William A. Barrett
John D. Couch

COMPILER CONSTRUCTION: THEORY AND PRACTICE

COMPILER CONSTRUCTION Theory and Practice

COMPILER CONSTRUCTION
Theory and Practice

WILLIAM A. BARRETT
JOHN D. COUCH

Composition — Typothetae and William A. Barrett
Acquisition Editor — Alan W. Lowe
Project Editor — Jay Schauer
Technical Art — Blakely Graphics

Printed in the United States of America.

Library of Congress Cataloging in Publication Data

Barrett, William A
Compiler construction.

Bibliography: p.
Includes index.
1. Compiling (Electronic computers) I. Couch, John D., joint author. II. Title.
QA76.6.B367 001.6'425 78–26183
ISBN 0–574–21160–8

10 9 8 7 6 5 4 3 2 1

FOREWORD *by Harold Stone*

Compiler Construction: Theory and Practice is exactly what the title promises. It is an excellent mix of the mathematical foundations of compilers and the practical considerations required in developing high quality compilers for commercial release. The level of discussion is suitable for college juniors and seniors. The material is readily digested because all of the mathematical prerequisites are included in the book, and they are exposed in a highly palatable fashion.

One of the strengths of the book is that algorithms are normally discussed in high-level Pascal-like language that brings out the structure and flow of the algorithms with great clarity in contrast to similar descriptions in flowchart language or automation operations used by some earlier texts in the area. The authors do a particularly commendable job on the practical aspects of code generation, code optimization, and syntax error handling, all three of which are still "black arts" by comparison to the topic of parsing where theory and practice have largely merged.

Here is a book that the professional compiler writer can use to create better language translators, that the computer scientist and engineer can use to gain an understanding and appreciation of the process of language translation that is part of his interface with the computer, and that the student can use to further his knowledge of the capabilities of computers and methods for harnessing the power of the computer.

FOREWORD *by W. M. McKeeman*

The art of translating programming languages has, in many ways, come of age. From its beginnings in Fortran and Algol and hundreds of other languages, we find the concept of high-level languages well established and the construction of translators routine.

Such progress exists because of the highly developed craft of compiler writers to whom the tools of the trade are well-known and regularly applied. There are few places where that rich craft is recorded altogether in a way that is understandable and immediately applicable. To provide such a source is the purpose served by the following text.

The reader must be a programmer. The terminology comes from that field, and the insights necessary to understand the material do so as well. Assuming that background, the reader should find here the world of automatic translation opening up—from the formalities of language description to the tough details of machine code generation. It is a fascinating subject and well worth the intellectual effort of study.

PREFACE

Compiler Construction: Theory and Practice is intended as a one- or two-semester course in the fundamentals of compiler construction and/or language translation. It is designed especially for students with some programming and computer systems background. A strong background in discrete mathematics is helpful, but not required.

This text treats:

- Grammars, trees, and parsing fundamentals.
- Finite-state automata—their relation to regular grammars and regular expressions, their systematic generation, their reduction to minimal form, their representations, and their application to compilers.
- Top-down parsing—principles, LL(k) grammars, LL(1) and recursive-descent parsers.
- Bottom-up parsing—principles, precedence parsers, and the LR(k) parser.
- Syntax-directed translation—principles and application to translation.
- Symbol tables and operations—principles, scope rules, type rules, static representation of PL/I, Cobol, and Pascal structures. Efficient name table access methods.
- Run-time machine models—to support recursive procedure calls, block structure parameter passing, data space allocation, and data access.
- Practical machine systems—a review of three commercial machine systems and their application as a target machine. Details of loader design, support of partial compilation, relocation, recursive calls, etc.
- Optimization—efficient register allocation, constant folding, recognition of common subexpressions, and introduction to data-flow analysis.
- Error recovery—recovering from syntax, scanner and semantic errors, with special attention paid to LR(k) recovery, use of forward move, and experimental studies of recovery.

The Authors

The authors have both academic and industrial experience in compiler construction. William Barrett taught courses in introductory computer systems and compiler construction at Lehigh University, Bethlehem, Pennsylvania,

before accepting a staff position in software development at Hewlett-Packard, Cupertino, California. He is currently responsible for a systems programming language. Its compiler incorporates a number of advanced features described in this textbook. John Couch taught compiler construction at San Jose State University for several years. Formerly responsible for systems software development at Hewlett-Packard, including several compiler projects, he is now Director of Product Development for Apple Computer Co., Cupertino, California.

As teachers, the authors believe that some language theory is an essential part of compiler construction. However, much of language theory is irrelevant to compiler construction. This text should help bridge the gap between theory and practice.

The authors also believe that a balance should be struck between parsing and code synthesis, and between top-down and bottom-up methods. Although most of the synthesis material in this text pertains to bottom-up parsing, both parsing approaches are given approximately equal treatment. Many of the synthesis considerations are applicable to either parsing approach.

Features

Along with basic core material, this text contains source material that has been collected in one book for the first time, and original material found nowhere else.

The first three chapters present a core treatment of grammars, trees, parsing, and finite state automata. We have included only enough language theory and automata theory to understand parsers and compilers. We have omitted such topics as Turing machines and computability. Enough automaton theory is presented to make it possible to develop the commonly used top-down and bottom-up parsers, and to prove that they perform as claimed. We believe that, with the formal descriptions, proofs, extensive discussions, and the many examples of machines and machine traces, a student will be able to understand how a parser works, and will also be able to understand professional literature in programming languages for additional information.

Chapter 3 discusses finite state automata, their relation to regular grammars and regular expressions, and their reduction to minimal form. As an application, lexical analysis is discussed briefly. A designer must be aware that some languages, such as Fortran, pose difficult lexical problems. Such problems are discussed in some detail.

The remaining chapters contain much original material. An extensive discussion of recursive descent parsers will be found in Chapter 5. We not only describe this commonly used parsing method, but also develop an automatic generator and the conditions under which the parser operates correctly.

The LR(k) parsing methods, which are recently being incorporated into some commercial compiler systems, are discussed in considerable detail in Chapter 6. Methods of reducing the size of the stored LR tables and of estimating the size of the tables are also given. Parser reductions based on special grammar properties are described in some detail.

The organization of a symbol table for single and multiple pass compilers with multiple block levels is given in Chapter 8. Efficient access methods are also described there. We show how to organize a symbol table for PL/I structures and Pascal types, and how to resolve partially specified names.

Run-time data structures are dealt with in Chapter 9. We develop a special machine system that can be used to implement Algol, Fortran, Pascal, and many other languages. This system, or its equivalent, must be emulated or simulated in order to support various features of these languages. The important issues of procedure calls, parameter passing, blocks, and array and structure access are dealt with in considerable detail. The material on the efficient access of multi-dimensional arrays and structures should be of particular interest.

Three specific machine architectures and their supporting system conventions are described in some detail in Chapter 10: the HP 3000, CDC 6400, and the IBM 360. The HP 3000 system is described at length. However, the principles of loader table design and machine architecture, exemplified in the HP 3000, are applicable to any system. In particular, we show how procedures can be independently compiled and brought together later by a loader to form a complete program.

Optimization issues are dealt with in Chapter 11. We have followed a modern school of thought in this area—that of constructing a directed acyclic graph of a program segment, then performing various reductions on that graph. We consider the multi-register allocation problem at some length, following the work of Aho and others in this field. Finally, we develop the fundamental notions of flow analysis, which can be used to reduce code and to detect subtle programmer errors during compilation.

The book ends with a chapter on error recovery. We classify program errors, then deal at some length with the problem of patching over a syntax error in a free-form language. We give different methods of dealing with syntax errors and discuss their effect on semantic issues. We summarize the results of some error recovery experiments.

Designing a Compiler Course

For a one-semester course, some portions of this text should be covered in detail; other sections should be approached tangentially. To concentrate on basic matters in one semester, Chapters 1–4 and 6–9 provide a solid grounding in grammars, top-down and bottom-up parsers, syntax-directed translation, symbol table issues, and static and dynamic data structures. We sug-

gest skipping Chapter 5, on precedence methods, and Chapters 10 through 12.

For a two-semester course, use Chapters 1–7 and Chapter 12 for the first semester, then 8–11 for the second semester. These chapters should be augmented with additional outside material or a class project. Many suggestions for projects are contained in exercises in these chapters. If the course is being taught for the first time, a useful project would be a top-down or bottom-up parser generator, based on the algorithms given in the text. This generator can then be used in subsequent years as a tool for practice in designing grammars and in writing compilers.

A simulator of the system described in Chapter 8 and a compiler that generates code for this system would also be instructive projects.

Acknowledgements

We wish to thank Frank DeRemer, Harold Stone, and Bill McKeeman for many helpful criticisms and a detailed reading of the manuscript; Richard Page of Hewlett-Packard for simulating the AOC machine and correcting potential errors; Steve Glanville for his criticism and for permission to use his thesis material; Tom Pennello for permission to use his thesis material; George Miller of the University of California for classroom testing this text; and Fred Clegg, and Tom Whitney for their patience, encouragement, and understanding.

We wish to thank Hewlett-Packard for allowing us to use their computer systems for manuscript preparation. We thank Control Data Corporation, IBM, and Hewlett-Packard for permission to create illustrations based on their technical manuals.

Finally, we thank our wives who supported us, despite the fact that we were around the house much less than they (and we) would have liked, through the many long hours spent working on "The Book."

W. B.
J. C.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Translators

A *translator* accepts a *source program* and transforms it into an *object program*. The source program is a member of a *source language* and the object program is a member of an *object language*. Both languages are artificial, inasmuch as they are designed for a digital computer, as opposed to a natural language like English or German.

Each program expresses some algorithm. We are primarily interested in those translations for which the source and object algorithms are identical. For example, a Fortran program should yield the same results for a given input regardless of the machine language to which it is translated. Those results should be as expected from the specification of the Fortran language and the algorithm as expressed in Fortran.

Artificial translation is rapidly becoming a mathematical discipline, while natural translation remains rather more an art. Yet the two are somewhat akin. Any student of foreign languages knows that one language cannot be translated to another by simply substituting words. A human translator must first grasp the precise meaning of each source sentence, then compose an equivalent sentence in the object language. So it is with artificial translators. A source program must first be analyzed to uncover its underlying meaning and structure; this process is called *parsing*. Then a number of transformations on the structure are performed, ultimately ending in the object program.

For a given source language, the translation may be carried to several different levels of completeness: to *assembly code*, to *machine code*, or to execution. Assembly code is a sequence of mnemonic instructions and symbolic address references; it is a member of an *assembly language*, and must be translated into machine code by yet another translator called an *assembler*. Assembly language usually has a very simple structure with a fixed format—a program location field, an instruction field, and an address field. Each line of assembly code usually translates to one machine instruction. There are no nested statements, arithmetic expressions, or procedures as in Fortran or Algol.

Machine code is a sequence of binary machine instructions that require little or no modification in order to be executed.

An *interpreter* accepts a source program, translates it into some intermediate data structure, then executes the algorithm by carrying out each operation given in the intermediate structure. An interpreter is considerably less efficient than a compiler, because it carries the burden of intermediate structure analysis as well as execution. However, a program can be rapidly

developed with an interpreter, since its test can follow its modification so rapidly.

The advantage of a compiler is that it generates an efficient, short, executable program. It demands fairly heavy computer resources while compiling, but when executing, only those resources needed by the executing program are required. The disadvantage of a compiler is the lag time between writing a program and executing it.

An interpreter as an algorithm is very similar to a compiler. The analysis of the source statements and bookkeeping for identifiers and literals are tasks common to interpreters and compilers. We shall therefore not be concerned with the differences between an interpreter and a compiler in this textbook.

A compiler or an interpreter is itself a program written in some language, called the *host* language. We therefore see that three languages are involved in a compiler– source, object, and host. These are often three different languages. A Fortran compiler that runs on an IBM 360 might be written in PL/I, and generate machine code for a 1401. A compiler that generates code for its host machine is called *self-resident*; if, in addition, it is written in its own source language, it is *self-compiling*. If it generates code for a machine other than the host, it is called a *cross compiler*.

1.2. Why Write a Compiler?

A programming language is designed and a compiler written for it for only one reason–to make it easier for human beings to get a computer to carry out a class of tasks. If it were possible for us to rapidly translate a task description into the long lists of binary numbers that a computer expects as its instruction list, with no errors, then programming languages and compilers would be unnecessary. Unfortunately, human beings make mistakes. They are unable to cope with a big list of binary numbers, and their time is valuable compared to machine time. A computer is well suited to clerical tasks and can handle them cheaply and accurately. One task that a computer can be expected to perform is assisting us in programming itself.

Here are some of the services that a compiler should provide for its user:

1. Evaluate symbolic references to instructions and instruction locations. Consider branches. At the machine level, a branch might look like this in 16-bit octal code:

location	contents
037663	024433

On an HP 2100 minicomputer, the number 024433 is interpreted as a direct branch to location 433. (The "024" is the instruction and the "433" is the address.)

Now it is unreasonable to expect anyone to remember the rather complicated pattern of bits that make up a branch instruction, nor those of the

dozens of other instructions offered on a typical small computer. It is therefore better to assign a mnemonic name such as JMP to each instruction. Then the same branch might be written

location	contents
037663	JMP 433

This change reads better than the 024433, yet now requires some kind of translator, one which can interpret the characters in "JMP" and turn them into the "024" of the JMP instruction on a 2100. The other instructions may similarly be given mnemonic codes. A simple program might then look like this:

	location	contents	comment
(start)	0	LDX 7	{Load index register, address 7}
	1	LDA 7,X	{Load accumulator, indexed}
	2	DSZ	{Decrement X and skip if zero}
	3	JMP 6	{Go to location 6}
	4	ADA 10,X	{Add location 10, indexed, to accumulator}
	5	JMP 2	{Go to location 2}
	6	HLT	{Stop}
	7	4	{Data needed by the program}
	10	16	
	11	32	
	12	176	
	13	24	

This program can now be read by someone familiar with the 2100 instruction set. The intention of the programmer was to add up the list of four numbers in locations 10 through 13. However, the program doesn't do that; it simply loads 24 in the accumulator register, then halts. There are several errors; a correct program is given below:

location	contents
0	LDX 6
1	LDA 7,X
2	ADA 6,X
3	DSZ
4	JMP 2
5	HLT
6	3
7	16
10	32
11	176
12	24

Now these errors would be obvious only to someone with considerable experience in 2100 assembly language coding. Yet this is quite a simple algorithm. How many errors are likely to be introduced in the assembly-language coding of a large data base manager or of an operating system?

Notice that lots of things have changed between the two programs. The most painful change is the removal of one instruction, which has caused a shift in the locations of the variables and all the instructions past the deleted one. In other words, every instruction referring to one of the shifted constants must be changed. One simple change in a long list of instructions can require many changes throughout the program.

The burden of finding and changing lots of instructions can be removed as a human activity and shifted to a computer by introducing symbolic location names. Any location that must be referred to in an instruction is assigned a symbolic name; then only names need appear in instructions. The assembler must then determine the location of each label and fix the instructions accordingly. The sample program then might look like this:

```
      LDX   L1
      LDA   L2,X
L4:   ADA   L1,X
      DSZ
      JMP   L4
      HLT
L1:    3
L2:   16
       32
      176
       24
```

Symbolic labels carry several unexpected bonuses. The absolute locations have disappeared, which means that our program may now be placed anywhere within some other program; the assembler will work out the locations. Only those locations that must be referenced are labeled; however, the label associated with some instruction may be hard to find in a large listing. The assembler may check that every symbol appears as a label exactly once in the program.

2. Constants should be converted to internal form by a compiler. Modern computers can handle a variety of internal data forms, for example, multiple-precision integers, floating-point (real) numbers, packed decimal numbers, strings, etc. No human being should be expected to perform the required conversion to internal form; there is also no valid reason why a programmer should even have to know the internal form.

3. Special control structures may be devised that read better than the primitive instructions of an assembler. For example, the sample program might read as follows in Algol:

```
INTEGER ARRAY A(4):=(16, 32, 176, 24);
INTEGER I, SUM;

SUM := 0;
FOR I := 0 UNTIL 3 DO SUM := SUM + A(I);
```

Although this is somewhat more wordy than the assembly language form, it is certainly easier to understand. The data variables are clearly declared separately from the algorithm that operates upon them, and the one-line FOR statement expresses the desired operation very clearly.

If a concise description of an algorithm is the most desirable feature of a programming language, then the language APL probably takes top honors. The above algorithm to add the four numbers 16, 32, 176, and 24 is written this way in APL:

```
SUM ← +/(16 32 176 24)
```

4. Programs written in any of several common high-level languages are often transportable. By this, we mean that a program written in, say, Fortran, can be compiled and executed with few changes on any of several different computers, despite differences in internal architecture and instruction set. When software is transportable, a computer user with a large program library may change computers without incurring a heavy software development cost. Programs written in assembly language for one computer are worthless on any other computer. Computers also become obsolete, and the assembly language programs written for them become worthless.

5. Assembly language programs are much more likely to contain subtle errors than the same programs written in a high-level language. There are several reasons for this: assembly language is hard to read except by an expert very well versed in the machine; assembly language programs are "unstructured," i.e., there are no control structures to guide coding and reading; it takes many more assembly language instructions to achieve the same effect as a suitable high-level language statement; and a machine instruction often has many subtle side-effects to trap an unsophisticated programmer, e.g., a condition code or carry bit may be set or cleared and an index register value may be altered. A modern high-level programming language and its compiler should protect the programmer from many such error-causing complications.

6. A high-level language lends itself to the division of the labor of a software task. A programming team can agree on the properties of certain procedures or macros, then go their separate ways to write and check out their part of the whole task. This principle also applies to a sophisticated assembler, but more agreements need to be reached on programming conventions among the team during the design phase.

7. A modern high-level language will engender good programming style. For example, an IF-THEN-ELSE construct forces a programmer to

consider both alternatives of an IF test; a failure to deal with one cries out from the printed page. In assembly language, it is often difficult to find the two alternatives of a branching test, and it is easy to obscure one of the two. The compiler can be made to check array bounds, so that during testing, any instance of an out-of-range index can be detected and subsequently analyzed. Good programming style also means that someone else can read and understand the program without a lot of analysis.

8. A number of high-level languages are rather simple in structure and can easily be learned by someone with little or no computer background. For example, Basic is used extensively in some grade schools. Such languages make computer services accessible to people who would otherwise never consider using them.

These are some of the major advantages provided by a high-level programming language and a compiler for it. There are many others. All these result in an appreciable increase in engineering efficiency in the writing, maintenance, and modification of software.

1.3. The Cost of a Compiler

The development of a compiler is a major software effort. Depending on the complexity of the language and the target machine, as little as three man-months or as much as thirty man-years may be required to write and debug a compiler. The most complex compiler ever written was probably PL/I for the IBM 360. PL/I is an extraordinarily rich language, containing not only several file access methods, but a large set of data types and operations.

Another cost is a certain loss of machine efficiency for a program written in a high-level language compared to the same algorithm written by a skilled programmer using assembly language. A high-level language imposes certain constraints upon a programmer in its forms of control structures, limited data types, etc., which do not exist in assembly language. This loss of efficiency is particularly severe for a high-level language that is not particularly well suited to its target machine. Thus Algol and PL/I are rather well suited to a stack architecture. A multiregister machine, such as the 360, generally requires elaborate optimization techniques in order that a compiler can compete with an assembler in number of words of code and execution time. The optimization phase of compiler design for a machine poorly suited to the language can double the compiler cost and size.

A compiler's inefficiency in generating executable code is paid for upon each use of the code. If that cost is deemed too high, several alternatives can be chosen, among them recoding in another language, or coding portions of the software in assembly language. Often an inefficiency in a programming system stems from a poor choice of algorithm, or a poor peripheral device access strategy, rather than from an inherent inefficiency in the compiler.

1.4. The Compiling Process

The major operations in a compiler are illustrated in figure 1.1. The process begins with a source file at the top of the figure, and ends with optimized object code at the bottom. Our description in this section will necessarily be highly simplified; many special problems in a real compiler system will be overlooked in this review.

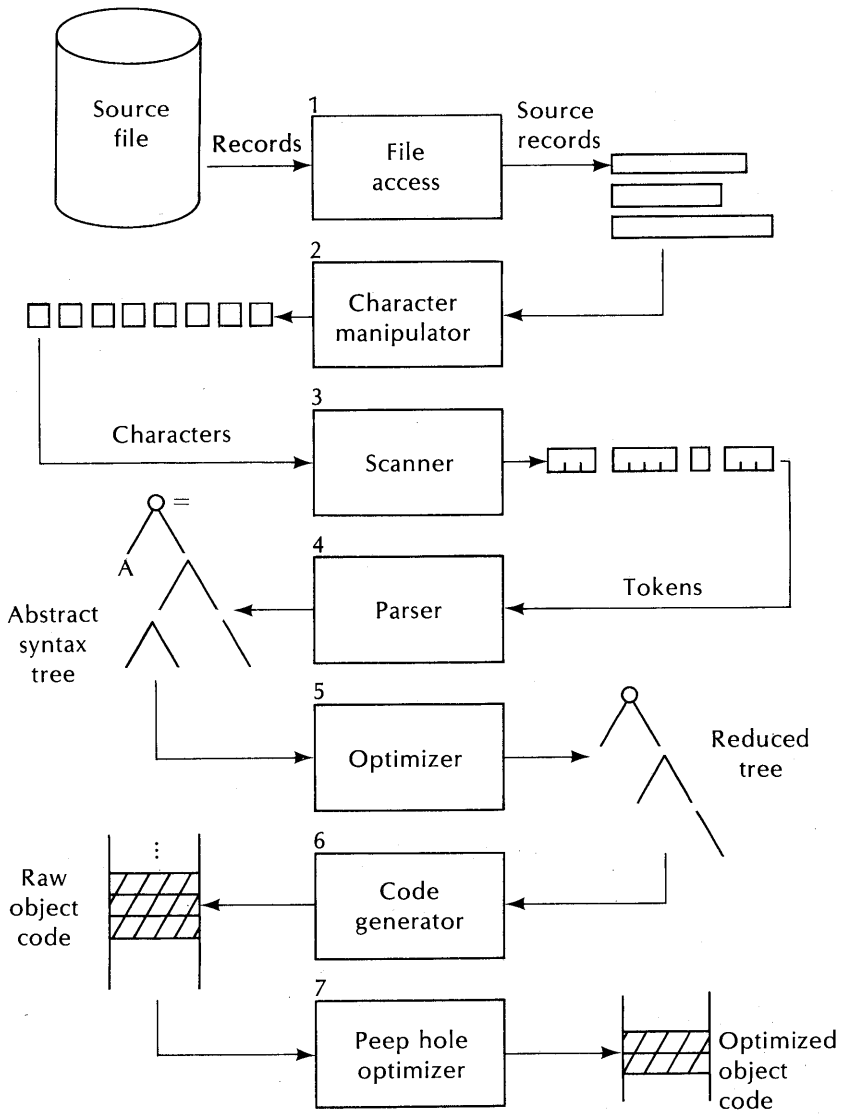


Figure 1.1. Major operations in a compiler.

A compiler is based on a sequence of transformations that preserve the operational meaning of a program, but not necessarily all the information in it, nor even the exact sequence of operations requested (or thought to be requested) in the source program.

The nature of the system upon which the compiler resides has a strong influence on the design of a compiler. Most computers have a severely limited "fast" memory (semiconductor or magnetic core), but extensive "mass" memory (disk, drum, magnetic tape). A compiler is often expected to process very large source programs, so that only a relatively small portion of the source can be actively under process at any one time. A compiler must usually be such that only the least information necessary be retained in fast memory, and such that most of the compilation is sequential in character–object code or some intermediate structures will be emitted as additional source is read.

Not all translators fit this pattern, nor must a compiler be strictly sequential on all systems. A system with virtual memory, for example, has unlimited memory available, in effect, so that sequential processing is less important. An interpreter usually has to carry all of the source, symbol tables, and program along during editing and execution; however, they need not necessarily all be in fast memory.

Compilation begins with some source form, shown in figure 1.1 as a file. Of course, source can originate in any of a variety of forms, such as punched cards, paper tape, a terminal, and magnetic tape. Their access is quite different, but the differences are usually of little or no concern to the compiler writer. File access (box 1) is generalized in most operating systems, so that the particular form of source is of no consequence. The compiler therefore first sees some sequence of source records, emitted by box 1.

In some languages, source record boundaries are important as statement delimiters (e.g., Fortran or Basic). In others, source boundaries are of no consequence. Hence a compiler will likely contain a section that accepts source records and emits a sequence of characters, box 2. This section may well detect and remove comments and special control commands that have nothing to do with the source language. If the language specification is such that blanks are ignored, then blanks would be suppressed by the character manipulator, box 2. However, this is not always easy. In Fortran, blanks are crucial in some contexts but not in others, and the necessary distinctions are sometimes difficult.

The character sequence is next subdivided into a sequence of *tokens* by a *scanner* or *screener*, box 3. A token may be a single character, or some special sequence of characters. The scanner may be responsible for skipping comments and blanks, if necessary. Examples of tokens are identifiers (names assigned to variables, statement labels, etc.), quoted strings, numbers, and special character sequences, such as ":=" in Algol.

Boxes 2 and 3 are sometimes called a *lexical analyzer*. They must be tailored to the language and the grammatical description of the language

chosen by the compiler implementors. For example, the Fortran source record

```
6      DOI       =4,X *( Y-16)       ,16
```

would be translated by a lexical analyzer into the token sequence:

```
6
DO
I
=
4
,
X
*
(
Y
-
16
)
,
16
```

The token sequence emitted by the lexical analyzer is next processed by a *parser*, box 4, whose task is to determine the underlying structure or "meaning" of the program. Until the parser is reached, the tokens have been collected with little or no regard to their position within the program as a whole. The parser considers the context of each token and classifies groups of tokens into larger entities such as *declarations*, *statements*, and *control structures*. The product of the parser is usually an *abstract syntax tree*. (An example of an abstract syntax tree for the Fortran DO statement is given in figure 1.2.) A tree is a very useful representation of a program or program segment, inasmuch as it facilitates several subsequent transformations designed to minimize the number of machine instructions needed to carry out the required operations.

The abstract syntax tree may next be subjected to a number of optimizations through a tree optimizer, box 5. The result is some reduced tree, possibly rearranged to suit the needs of the machine architecture. Examples of optimization possible at this level include: constant expression evaluation, use of commutativity, associativity, and distributivity of certain operators to collect constant expressions together, detection of common subexpressions, or rearrangement to suit a particular architecture.

The reduced tree may then be transformed into a sequence of raw object code by a code generator, box 6. The object code may be of several different

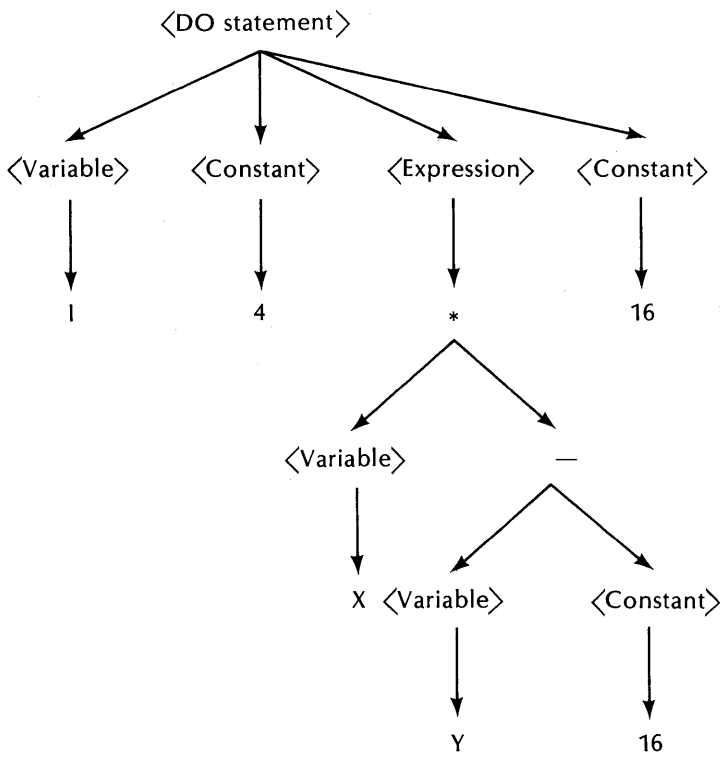


Figure 1.2. Abstract syntax tree for the Fortran statement DO I=4, X*(Y−16), 16

kinds, depending on the purpose of the compiler: (1) it may be machine code for some particular target machine, or (2) it may be a special intermediate file designed to be further processed by a loader, or (3) it may be specially designed intermediate code that must be further translated by another system into machine code or a loader structure.

Finally, the raw object code may be subjected to further optimizations by a *peep-hole optimizer*, box 7. Such an optimizer examines short sequences of code and determines whether, in certain cases, a sequence can be replaced by a shorter equivalent sequence. For example, the sequence

```
STOR  A;  {copy accumulator to location A}
LOAD  A;  {copy location to accumulator}
```

could be reduced by removing the LOAD instruction. This sequence can easily occur if the STOR represents the end of one statement and LOAD the beginning of another. As another example, the sequence

```
LOAD  A;
ADD   =1;{add constant 1 to accumulator}
STOR  A;
```

might be replaced by a single instruction,

```
INCM  A;  {increment contents of location A}
```

on a machine containing such an instruction. Such optimizations can appreciably reduce the number of emitted instructions.

Other operations not shown in figure 1.1 include *string table management* and *error recovery*. The string table contains a copy of each identifier appearing in the program. It is usually linked to an *attribute table*, so that properties assigned to some identifier can be made available throughout the compilation process. The string table and attribute table together are called a *symbol table*. The attributes of an identifier might include:

- How it was declared, (e.g. REAL, ARRAY).
- Its dimensions, if an array.
- The number of storage elements required for it.
- Its location in memory.

An *error recovery* system is usually attached to the parser, box 4, and takes control when a syntax error in the source program is detected. Its purpose is to diagnose the error and attempt some kind of recovery, so that the subsequent source input need not be discarded. The error recovery system may, for example, decide to insert some tokens, or discard some input tokens in an attempt to patch over the offending section of source.

1.5. Translator Issues

Often a very simple compiler can be made by omitting all the optimization steps, even the tree-building step. Many compilers generate machine code directly through the parse steps, using a variety of heuristic methods to achieve the translation.

A compiler might translate a source language to another closely-related intermediate language, or to symbolic assembly code. The latter can then be translated by some existing compiler or assembler to machine code. Many of the functions of a full compiler can thus be omitted, considerably reducing the compiler development task. However, such a compiler will likely neither be very efficient nor generate particularly efficient object code.

A compiler that generates an intermediate language could be used with several different machines. Of course, a translator from intermediate lan-

guage to each machine's code is also needed, but these are often easier to write than several different complete compilers. For example, a translator from Pascal to PCODE has been used to implement Pascal on several different machines. In some of the implementations, PCODE is interpreted, rather than translated.

The steps outlined in figure 1.1 may be carried out in one or several *passes*. A pass in general is some scan through either the source records, or through some translation of the source. A practical multipass compiler requires sufficient temporary memory to hold the intermediate translation; this could be any read/write medium. The intermediate structures may well be much larger than the final machine code, and in any case, during compilation, the machine's memory must be shared with the compiler program and the compiler's data areas.

In a one-pass compiler, the different sections of the compiler represented in figure 1.1 appear as different procedures in one large compiler program. Whenever a point in some process is complete, a procedure may be called that accomplishes the next step. For example, the parser may be the primary "driver" for the compiler, i.e., the main program, first called. It might then call the lexical analyzer to deliver one token; the lexical analyzer in turn may call a file handler to deliver one record, etc. The parser in turn might then call a tree-builder as the various parts of a source statement are analyzed. The tree could be built in one of several different ways, but when enough is completed, a tree optimizer might be called, then it calls the code emitters. In such a system, all of the steps of figure 1.1 are repeatedly performed as the compiler scans the source text, and code is emitted in short segments that correspond to segments of source text.

In a multipass compiler, each of the steps in figure 1.1 might be a separate pass. In practice, some groups of steps are combined, for example, it is feasible to construct a sequence of abstract syntax trees in one pass, then devote subsequent passes to reducing or transforming the tree. The tree itself can either be stored as a linked-list data structure, or represented in some linear form, such as reverse Polish.

The principle advantage of a multipass compiler is its ability to collect information that can be used to efficiently allocate storage for variables and emit instructions, information that is often difficult to obtain in one pass. Some optimizations require several scans over a major source program module. The principle disadvantage is that it is only applicable to a computer system with sufficient intermediate storage. Small computer systems that have very poor or slow intermediate storage (e.g. paper tape) are therefore poor candidates for a multipass compiler.

Summary

A compiler, as a translator of one language to another, can be organized in a variety of ways, depending on the source language, the target language, the

degree of optimization desired, the time available to develop the compiler, and its future value. A characteristic of nearly every compiler is a means of translating source records into a sequence of tokens (characteristic of the language), parsing this sequence to yield a syntax tree, and then transforming the tree to yield the object program. Alternatively, it is usually possible to bypass tree building, and simply emit code directly from the parsing system. Modern practice seems to favor tree construction and multiple passes, because they facilitate a number of optimizations, modularize the compiler design, and are often more efficient than the alternatives.

For Discussion

1. Choose some assembler you are familiar with and make a list of the services that it provides its users. How valuable are each of these services? Are there services that it could provide, but fails to?

2. Consider two or three high-level programming languages that you are familiar with and discuss those features that you feel are most valuable to a programmer. Also discuss features that you would most like to see added to the language.

3. How desirable is brevity in a programming language? (Compare Cobol and APL, for example.) Discuss the merits and demerits of extreme brevity.

4. What do you feel should be uppermost in the design of a "good" programming language among the following characteristics? How much does your choice depend on the application or the user community? Which desirable features are likely to conflict with each other?

(a) Ease of grasping the program's algorithm, upon reading the program.

(b) Ease of learning the programming language for the first time.

(c) Protection against coding errors, to the degree possible in a language.

(d) Ability to use any feature supported by the target machine, as needed, in order to obtain the most efficient execution.

(e) Convenience to the keypunch or terminal operator.

(f) The number of different operations available in the language.

(g) Immediate line-by-line syntax checking while preparing the program.

(h) A large number of operations.

(i) Conciseness of expression. (Compare APL and Cobol.)

5. Suppose that you are given an assignment of writing a compiler for some language to be implemented on a machine equipped with a symbolic assembler and a Fortran compiler. Discuss in general terms your strategy, given each of the following objectives:

(a) The compiler may implement only selected features of the language L (your choice) and may be very inefficient, but must be finished as soon as possible.

(b) The compiler must implement the complete language, may generate inefficient code, but must run as efficiently as possible and require as little memory space as possible. (It will be used for short student programs that will likely be executed only once).

(c) The compiler may be as large as you have time and patience to develop it and may be inefficient as a program, but must generate the most efficient code possible for the target machine.

(d) The compiler may generate inefficient code and be inefficient as a program, but must later be transformed into a compiler for another target machine with minimum effort.

CHAPTER 2

INTRODUCTION TO LANGUAGE THEORY

2.1. Language Elements

The elements of a language are its *alphabet*, its *grammar*, and its *semantics*. The *alphabet* is a finite set of *tokens* of which sentences in the language are composed. The *grammar* is a set of structural rules defining the legal contexts of tokens in sentences. The *semantics* is a set of rules that define the operational effect of any program written in the language when translated and executed on some machine. This and the next four chapters deal principally with grammars and translation apart from semantics.

Sentences in English are constructed from the set of characters consisting of the letters, digits, space, and punctuation marks. These characters are composed into words through the aid of spelling rules and a dictionary, and then words are composed into sentences through the aid of grammatical rules.

A computer program may similarly be constructed from a sequence of characters drawn from the computer's character set. Such a sequence may be composed into tokens (corresponding to English words) and the tokens composed into sentences through a grammar. The principal difference between English and a programming language is that the grammatical and spelling rules for English are very complicated and have many exceptions and ambiguities, while the corresponding rules for a programming language are concise, highly structured, have few (if any) special cases, and (hopefully) no amibiguities.

In this chapter, we will develop the notions of alphabets, strings in an alphabet, generative grammar systems, and the problem of *parsing* a sentence. Parsing is the process by which we determine the specific grammar rule applications that yield a given sentence; it corresponds roughly to determining the structure of an English sentence.

2.1.1.Tokens and Alphabets

An *alphabet* is a finite set of *tokens*, fixed at the time of definition of the source language. Every source program consists of some sequence of tokens drawn from the alphabet of the source language.

The alphabet of a programming language could be a set of keyboard characters; each letter, each blank, each digit, and special symbol is a distinct token, and it is possible to define useful programming languages on such an alphabet.

More often, certain easily recognizable sequences of characters are collected together to form the tokens of the language. For example, in Algol 60, the characters ":" and "=" when written together in that order ":=" represent one token which is used for assignments. In Fortran, the character sequence DO usually represents a special token that initiates a loop structure. The task of assembling a sequence of source characters into tokens comprises that part of a compiler called the *scanner* or *lexical analyzer*. That task may be simple or difficult; the end result in any case is to yield a sequence of tokens for the benefit of the language structural analysis that is to follow.

Examples of alphabets:

1. The Fortran character set, which contains the 26 letters, the ten digits, and the special symbols $+ \; - \; * \; / \; . \; , \; (\;) \; = \; \$ \; ' \; :$, a total of 48 characters. Every Fortran program may therefore be considered as one long string of characters. For practical reasons, the end of a Fortran statement and a blank may also be considered characters, and are important as language elements. The end of a Fortran program, or end-of-file, may also be considered a character in the alphabet, bringing the alphabet to 51 characters.

2. The set of Fortran special names, identifiers, constants, and special symbols. In this alphabet, special names such as DO, IF, and READ are separate tokens, distinct from names invented by the programmer. Similarly, a number is considered a token. A name invented by a programmer (an identifier) is considered an element of the alphabet.

The symbol Σ will be used to designate an alphabet. Mathematically, Σ is a set of objects. Lower case letters near the beginning of the English alphabet, e.g., a, b, c, d, will usually represent members of Σ.

2.1.2. Strings

A *string* is a sequence of elements drawn from an alphabet. The set of all strings of length one or more consisting of members of Σ will be designated Σ^+. Thus if an alphabet Σ consists of the characters #, 4, %, and +, written:

$$\Sigma = \{\#, 4, \%, +\}$$

then each of these strings is a member of Σ^+:

#

4444# #%+

+# #4

+++

There are, of course, many other possible members.

The length of a string is the number of characters in the string, written:

$$|\text{FORTRAN}| = \text{length of the string "FORTRAN"} = 7$$

Thus $|4444\#\ \#\%+|$ is 8, $|\#|$ is 1, etc.

Lowercase letters near the end of the alphabet, e.g. w, x, y, will be used to represent strings.

A string whose length is zero is called the *empty string*, written ϵ.

Two strings, w and x, may be connected together to form a single new string, wx. This operation is called *concatenation*. Either of the strings may consist of a single character, multiple characters, or the empty string. Concatenation is an implied operation and as such has no special symbol. The concatenation of an empty string with any other string x results in x; that is:

$$\epsilon x = x$$

$$x\epsilon = x$$

$$x\epsilon y = xy$$

Note that if strings x and y are each members of Σ^+, then the string xy is also a member of Σ^+.

Although the empty string "disappears" when concatenated with any other string, it could be a member of any set of strings. In particular, the set Σ^* is the set Σ^+ together with the empty string, which can be written using the set union operator $\cup$ as:

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

In terms of compilers, an empty string corresponds to a source program containing no tokens. In practice, such a program must be followed by an end-of-file indicator. The compiler then can conclude that the program is an empty string by detecting the end-of-file as the first token.

An empty string is not the same as an empty set $\varnothing$. A set may consist only of the empty string; such a set is not empty. An empty set contains nothing, not even an empty string. Again, there is a correspondence of these concepts in a compiler. A compiler may accept an empty string by noticing an end-of-file marker. A compiler that accepts an empty set will reject every input including an empty input and will then halt regardless of the input source.

2.2. Generative Grammars and Languages

A *language* is some subset of Σ^*, where Σ is its alphabet. This definition is much too broad to be of any practical use. If the alphabet consists of the English characters, then Σ^* contains the text of all the books in the world, but

also contains whatever happens to be pecked out by a monkey at a typewriter. Σ^* includes meaningful sentences of value to a reader, but also includes random sequences of characters.

A language may also be finite or infinite. If a language is infinite, there cannot be a bound on the maximum length of a string in the language. If there were such a bound, then because the alphabet is finite, there is a finite number of arrangements of the tokens in the finite-length strings.

To be useful, a language must contain structure. There must be a reasonably small set of rules that govern the manner in which the tokens of Σ may be organized into strings in the language. We call the set of rules that define a legal class of strings a *grammar*.

For example, telephone numbers in the United States have a certain structure familiar to everyone—an area code, an office number, a party number, and an extension, for example:

(212) 438-7021 X643

Some of these components may not appear in a phone number; if there are no extensions or the area code is understood, they may be omitted. Also some telephone systems permit dialing only the party number. We can represent a phone number by the structure

<area code> <office> <party> <extension>

Each of these has a structure of its own. The area code is commonly written in parentheses, e.g., (212), so we see that the structure <area code> has the structure

(<three-digit number>)

and in turn, this structure <three-digit number> has the structure

<digit> <digit> <digit>

Finally, each <digit> has one of the forms 0, 1, 2, . . . , 9.

The overall structure of a phone number can be seen as a set of structures, each of which provides a more detailed description of the number. The "smallest" components are the members of the phone number alphabet, consisting of the digits, dashes, parentheses, and "X".

2.2.1. Terminals and nonterminals

A *terminal* is any member of Σ, and is therefore a synonym for token. A *nonterminal* stands for some set of strings in Σ^* , but is not itself in Σ. Nonterminals are used in a language's structural rules. For a given grammar, there will be a finite set of nonterminals; this set will usually be designated N. A *symbol* is a terminal or a nonterminal.

In the telephone number structure example given above, each of the names

<area code>
<office>
<party>
<extension>

are nonterminals. We could equally well use single characters as nonterminals, and shall do so frequently. For most of the simple example grammars in this text, we will use capital letters near the beginning of the English alphabet to designate a particular nonterminal, e.g. A, B, C. A capital letter near the end of the English alphabet, e.g. X, Y, Z, will generally stand for an "arbitrary" nonterminal.

For large grammars, we will need more than 26 nonterminals, and will therefore fabricate special nonterminal names. The convention of using < > to designate a nonterminal is used widely to define programming languages. The nonterminals <area code> and <office> are examples of this convention.

2.2.2. Production Rules and Grammars

A *production rule*, or *production* for short, has the general form

$$x \rightarrow y$$

where x and y are strings in the terminal and nonterminal sets of a given grammar. The y may the empty string, but x cannot be. Productions are used in a special subclass of grammars called *replacement systems*. Essentially, in such a system, we can generate other strings in the language by starting with a string of the form

wxz

then applying a production of the form $x \rightarrow y$ to yield a new string

wyz

We have effectively replaced x in the string wxz by y, through a given production $x \rightarrow y$. The y is called a *simple phrase* of wyz. The replacement step wxz to wyz is called a *derivation step*, and we say that wxz *derives* wyz.

Eventually, given enough such replacements and productions, we could end up with a string of terminal symbols. However, we haven't yet specified where wxz came from, nor have we restricted the productions $x \rightarrow y$ in any way.

If we could start with any string at all, and then commence with replacement rules, we would really have failed to define a language. One possibility is "no replacement," so such a strategy leads to a language

consisting of Σ^* again. We therefore need some uniquely specified starting string. We may in fact choose one nonterminal as the starting string, rather than some other string w, without loss of generality, since we can always add a production of the form $S \rightarrow w$ to a grammar whose starting string is w.

Next we need some precise way of knowing when to stop a sequence of derivations. We choose to stop when we obtain a string that contains no nonterminals; the nonterminals after all are subject to further reduction. This feature implies two other important properties that a grammar should satisfy: (1) the nonterminal and terminal sets are disjoint. If a token is both terminal and nonterminal, it is not clear whether to stop the derivation or not. (2) Every production rule must contain at least one nonterminal in its left member, i.e., given a production $y \rightarrow x$, y must contain at least one nonterminal. If both properties are satisfied by the grammar, then no further derivations are possible once we have an all-terminal string.

We conclude that a grammar is a four-tuple (Σ, N, P, S), where Σ is a terminal alphabet, N is a nonterminal alphabet, Σ and N are disjoint, P is a set of productions of the form $y \rightarrow x$, where y and x are in $(N \cup \Sigma)^*$, and y contains at least one element in N, and S is a designated start symbol in N.

Such a grammar is called a *phrase-structure* grammar.

2.2.3. Classes of Grammars

Chomsky [1965] distinguished four general classes of grammars. The most general class, the *unrestricted* grammars, is not phrase-structured. The other three classes are phrase-structured. They are the *context-sensitive*, *context-free* and *right-linear* grammars.

The most general phrase-structured class is the *context-sensitive* grammar. In this class, each production has the form

$$x \rightarrow y$$

where x and y are members of $(N \cup \Sigma)^*$, x contains at least one member of N, and $|x| \leq |y|$. Note that the latter requirement implies that y cannot be empty. An example of a context-sensitive grammar is

$$G_1 = (\{S, B, C\}, \{a, b, c\}, P, S)$$

where the productions P are

1. $S \rightarrow aSBC$
2. $S \rightarrow abC$
3. $CB \rightarrow BC$
4. $bB \rightarrow bb$
5. $bC \rightarrow bc$
6. $cC \rightarrow cc$

Let us develop a set of replacements in this grammar. Since S is the designated starting string, we look for a production with S as its left member. Either of the first two will do:

$$S \rightarrow aSBC$$

so that aSBC is a new string. Now in string aSBC, we can only use another S rule; let's choose the second one:

aSBC becomes aabCBC

Here, the third or fifth rule may be chosen; let us choose the third:

aabCBC becomes aabBCC

Continuing, we find the following sequence of replacements:

aabbCC

aabbcC

aabbcc

We end up with all terminals, so this is the end of the possible replacements.

We could reach a string for which no production can apply. For example, in the string aabCBC, if we choose the fifth rule instead of the third, we obtain aabcBC, and we find that no rule can be applied to this string. The consequence of such a failure to obtain a terminal string is simply that we must try other possibilities until we find those that yield terminal strings.

Exercises

1. Derive the strings

   abc
   aaabbbccc

   in G_1: $(\{S, B, C\}, \{a, b, c\}, P, S)$, where $P = \{S \rightarrow aSBC, S \rightarrow abC, CB \rightarrow BC, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}$.

2. Show informally that the strings

   abbc
   aabc
   abcc

   cannot be derived in G_1.

3. From Exercises 1 and 2, frame a conjecture regarding the kind of strings derivable in G_1, and attempt an informal proof of your conjecture.

4. Derive a context-sensitive grammar for strings of 0's and 1's such that the number of 0's and 1's is equal.

Context-Free Grammars

The next most general class of grammars is one that we shall be studying in most of this text—the class of context-free grammars. In a context-free grammar, or CFG, each production has the form $x \rightarrow y$, where x is a member of N, and y is any string in $(N \cup \Sigma)^*$. Note that y may be the empty string, hence any context-free grammar with a rule $A \rightarrow \epsilon$ cannot be context-sensitive; the latter class does not permit such a rule.

An example of a context-free grammar is one that we shall be using repeatedly, an arithmetic expression grammar G_0:

$$N = \{E, T, F\}, \Sigma = \{+, *, (,), a\}, S = E, \text{ and } P = \text{the set}$$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

Here, the nonterminal set is clearly {E, T, F}, the terminal set is {+, *, (,), a}, and the start symbol is E. We may obtain a typical expression by applying the replacement rules, as before:

E derives E + T, using the first rule
E + T derives T + T, using the second rule
T + T derives F + T, using the fourth rule
F + T derives a + T, using the last rule
a + T derives a + F, using the fourth rule
a + F derives a + a, using the last rule

Many other examples of derived terminal strings in this grammar may be obtained.

Whenever a grammar has several productions with the same left member, we will sometimes use the symbol "|" which stands for *alternation*. Thus the two rules $E \rightarrow E+T$ and $E \rightarrow T$ may also be written

$$E \rightarrow E+T \mid T$$

Exercises

1. Derive the following strings in G_0:

   a*a+a
   (a+a)*a
   ((a))
   a*a*a

2. Show informally that the following strings cannot be derived in G_0:

   a*+a
   a+aa
   ((a)

3. Show informally that in G_0:

   (a) Any string of the form (...(a)...) can be derived, where the number of left parentheses is equal to the number of right parentheses.

   (b) Any string of the form a+a+a+a+...+a can be derived.

   (c) The pairs *+, ++, **, and +* can never appear in a derivation.

   (d) The pair aa can never appear in a derivation.

4. Show informally that each of the nonterminals in G_0 can derive an arbitrarily long terminal string.

5. Give an example of a simple grammar containing a nonterminal that cannot derive any terminal string, i.e., where every derived string contains some nonterminal.

6. Construct a context-free grammar for telephone numbers, along the line introduced in section 2.2.

7. Write a context-free grammar for the following language. The notation 1^n stands for a sequence of n 1's.

   $$01^n01^n0 \text{ where } n \geq 0$$

   Examples of strings in this language are: 000, 01010, 0110110.

8. Write a context-free grammar that specifies the set of decimal literals that may be written in Fortran. Examples of these literals are

   −21.5
   0.25
   3.7E−6
   .5E7
   6E6
   100.E+3

   Note that E or a decimal point is sufficient to specify a decimal number. If neither is present, then the number is considered fixed point.

9. Given the grammar G = ({A, B, C, D}, {x, y}, P, A), where P is the set of productions

   A → B | D
   B → BCC | x
   C → yx
   D → xCyD | xy

   show that x, xy, xyxyx, and xyxyxyxyxy but none of xyx, xyxy, and xyxyxyx are derivable from A.

Right-Linear Grammars

If each production in P has the form A → xB or A → x, where A and B are in N and x is in Σ^*, the grammar is said to be *right-linear.*

The right-linear grammars are clearly a subset of the context-free grammars. An example of a right-linear grammar is the following grammar G_2; it defines a set of ternary fixed point numbers, with an optional plus or minus sign:

V → N | +N | −N
N → 0 | 1 | 2
N → 0N | 1N | 2N

Two other grammars related to the right-linear grammar are the *left-linear* and the *regular* grammar. A left-linear grammar has productions in P of the form A → Bx or A → x, where A, B, and x have the above meanings. A regular grammar is such that every production in P, with the exception of S → ϵ (S is the start symbol) is of the form A → aB or A → a, where a is in Σ. Further, if S → ϵ is in the grammar, then S does not appear on the right of any production.

An example of a regular grammar here defines the fixed point decimal numbers with a decimal point. The "d" stands for a decimal digit:

$$
\begin{aligned}
S &\rightarrow dB \mid +A \mid -A \mid .G \\
A &\rightarrow dB \mid .G \\
B &\rightarrow dB \mid .H \mid d \\
G &\rightarrow dH \\
H &\rightarrow dH \mid d
\end{aligned}
$$

Exercises

1. Derive the following strings in G_2:

   220
   −12
   +2

2. Show informally that the following strings cannot be derived in G_2:

   +
   2−0
   3+

3. Show informally that G_2 can derive arbitrarily long strings. What property of the grammar makes this possible?

4. Give a left-linear grammar that expresses the same set of terminal strings as G_2.

Significance of the Grammar Classification

These grammar classifications are to some extent arbitrary. One may define many variations on the basic patterns given. However, these particular definitions lead to particularly simple classes of sentence recognizing machines or *automata*.

An *automaton*, for our purposes, is some system with a finite description (but not necessarily containing a finite number of parts) that can accept some string of terminal symbols, given a grammar, and that can determine whether the string can be derived in the grammar.

The process of finding a derivation, given a grammar and a terminal string supposedly derivable in the grammar, is called *parsing*, and an automaton capable of parsing is called a *parser*. A parsing automaton is clearly of value in a compiler. A grammar is a concise, yet accurate description of some

language; it expresses the class of structures that we want in the language. However, so far we see only how to construct legal strings in the language. We need to solve the opposite problem: given some string, to determine if it is legal. We also need to go farther than that; we must determine the sequence of productions needed to obtain the string.

Now each of the three phrase-structured grammar classes has a fairly simple yet powerful automaton associated with it:

1. The right-linear grammars can be recognized by a finite-state automaton, which consists merely of a finite set of states and a set of transitions between pairs of states. Each transition is associated with some terminal symbol. We shall define finite-state automata more completely in the next chapter.

2. The context-free grammars are accepted by a finite-state automaton controlling a push-down stack, with certain simple rules governing the operations. The push-down stack is the only element that can be indefinitely large. However, only a finite group of top stack members are ever referenced in the finite description of this automaton.

3. The context-sensitive grammars are accepted by a two-way, linear bounded automaton, which is essentially a Turing machine whose tape is not permitted to grow longer than the input string.

Of these three, only the first two will be dealt with at length in this textbook. It turns out that the class of context-free grammars is sufficiently powerful to encompass most of the features of nearly every common programming language. Those features which are not covered by a context-free grammar are not in practice covered by a context-sensitive grammar, either, but require special extensions to the recognizing automaton.

2.2.4. Sentential Forms and Language Definition

Recall that in one derivation step, we transform a string

$$wxz$$

into a string

$$wyz$$

given a production $x \longrightarrow y$ in the grammar. We represent a derivation step by the symbol $\Rightarrow$,

$$wxz \Rightarrow wyz$$

Each of the strings wxz and wyz are called *sentential forms*, provided that we started with the start symbol S of the grammar and obtained wxz through a sequence of derivation steps.

A sequence of one or more derivation steps is indicated by

$$\Rightarrow^+$$

For example, in grammar G_0, we have

$$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow T+T+T$$

hence we may write

$$E \Rightarrow^+ T+T+T$$

Here, since E is the start symbol, the string $T+T+T$ is a sentential form in G_0.

A sentential form consisting only of terminals is called a *sentence*. Clearly, if we have

$$S \Rightarrow^+ (\text{some sentence})$$

then we have obtained, or *derived* a member of the language defined by the grammar. We may put this in set notation as follows.

Given a grammar $G = (N, \Sigma, P, S)$, then L(G), the language defined by G, is

$$L(G) = \{x \mid x \in \Sigma^* \text{, and } S \Rightarrow^+ x\}$$

The form $\{x \mid C\}$ is read: "the set of all x's such that condition C holds." L(G) is then the set of all strings x such that x is terminal and x can be derived from the start symbol S.

This definition holds for any of the three language classes context-sensitive, context-free, and right-linear.

Under this definition, L(G) may consist of the null set, or it may consist of one or more terminal strings, possibly including the empty string. For example, consider the grammars G_E and G_O, as follows:

$$G_E = (\{S\}, \{\epsilon\}, \{S \rightarrow \epsilon\}, S)$$

$$G_O = (\{S\}, \{\epsilon\}, \{S \rightarrow S\}, S)$$

In neither case is the production set empty (it could be, incidentally). However, in the first case, we have a language consisting of one string, the empty string. Such a language might be accepted by a compiler that notices that a source program contains nothing, and then halts (or in practice, goes on to the next source program in a job sequence).

In the second case, G_O, the only production just yields another S and can never yield a terminal string. Nor can it yield the empty string; we need some way for $S \Rightarrow^+ \epsilon$ to do that. We conclude that the language of G_O is empty ($\varnothing$), and is different from the language of G_E.

Sometimes it is useful to refer to a derivation of "zero" steps. This just means "no step"; the sentential form is left unchanged. We indicate a derivation of zero or more steps by the symbol $\Rightarrow^*$.

Exercises

1. Show that the following are sentential forms in G_0:

   (E+a)
   a+T*a
   (E)*a

2. Given a grammar with an empty production set, is its language empty?

3. Can a grammar have an empty nonterminal set? An empty terminal set? A terminal set consisting only of the empty string? If so, what can be said of the grammar's language in each case?

2.2.5. Production Trees and Syntax Trees

Recall that a tree is useful as an intermediate representation of a program or a portion of a program. It is also useful as a means of representing a derivation of some sentence in a context-free language, or of representing the productions of a CFG.

A *tree* is an abstract representation of a certain connectedness among a set of objects. The objects are called *nodes* and the connections among them are called *directed edges*. A tree may be constructed of distinct objects by the following recursive process:

1. One distinguished node is called the *start node*. Let N designate a start node; N is a tree.

2. Given any node N of a tree T with no out-directed edges. We may construct another tree T′ from T by adding one or more nodes $N_1, N_2, \ldots, N_n$ (not already in T) to T, and connecting each of these to node N by an edge directed from N to the node. The nodes $N_1, N_2, \ldots, N_n$ are called the *children* or *immediate descendants* of N, and N is called the *parent* or *immediate ancestor* of the nodes $N_1, N_2, \ldots, N_n$. The nodes $N_1, \ldots, N_n$ are *siblings* of each other.

Every tree has exactly one node with no indirected edges called the *root* node. A node with no outdirected edge is called a *leaf* or *terminal* node. Every tree has at least one terminal node that may also be the root. A node with at least one outdirected edge is called an *internal* node. A *path* is any set of nodes $n_1, n_2, \ldots, n_k$ such that one edge connects n_i to n_{i+1}, in that order, for all i such that $1 \leq i < k$.

The length of some path containing n nodes is $n-1$. For the sake of generality, we consider one node as a path of length zero.

Given any node N, there is a unique path from the root to that node. If its length is L, then node N is said to be at *level* L in the tree.

There is no path that connects a node to itself. A tree is said to be *acyclic* for this reason, however, there are acyclic graphs that are not trees.

The *height* of a tree is the maximum level in the tree, hence the length of the longest path. This longest path must extend from the root to some leaf node.

We will generally draw our trees upside down, with the root node at the top. figure 2.1 shows a tree with its parts labeled.

A tree may be embedded in a plane with each node a distinct point and no two edges crossing. The tree definition suggests how this can be done.

Each node N of a tree is the root of another tree, sometimes called a *subtree* rooted in N.

A tree embedded in a plane can be ordered in several different ways. The scheme we shall most often use is called a *preorder*, or *left-to-right natural order*, illustrated in figure 2.2. We obtain a preorder by imagining the tree surrounded by a directed circle, with the direction counterclockwise (figure 2.3). Then let the circle collapse around the tree, so that by following its path we contact every edge twice, once on one side and once on the other. We then order the nodes by assigning 1 to the root and the successive integers to the nodes when first touched by the collapsed circle.

A *postorder* is obtained by following the same procedure as a preorder, except that successive integers are assigned to the nodes when *last* touched by the collapsed circle.

Any tree can be represented in a linear memory space by a set of nodes, each of which contains two pointers: to its child and to its right sibling. Since either or both of these may not exist, a special pointer called a *nil* pointer is needed, that indicates this fact. Such a pointer system also imposes an ordering on the children. However, it is difficult to find the parent node of a given tree node N with this system; it is necessary to examine all the nodes in the tree starting with the root until we find that node one of whose children is N.

In order to locate a parent node rapidly, we may either add another pointer to each node, pointing to its parent, or set the right sibling pointer of each right-most sibling to point to its parent. The latter kind of tree is shown in figure 2.4. We need only some mark on the right-most sibling to indicate that its right sibling pointer points to the parent, not to its right sibling.

A Pascal data structure for such a tree is the following:

```
type TREENODE: record CHILD, SIBLING: ↑ TREENODE;
                      PARENT: Boolean
               end
```

The symbol ↑ means that CHILD and SIBLING are pointers to a data structure of type TREENODE. If PARENT is TRUE, then SIBLING is the right-most sibling, and it points to its parent. The links in figure 2.4 clearly are in preorder.

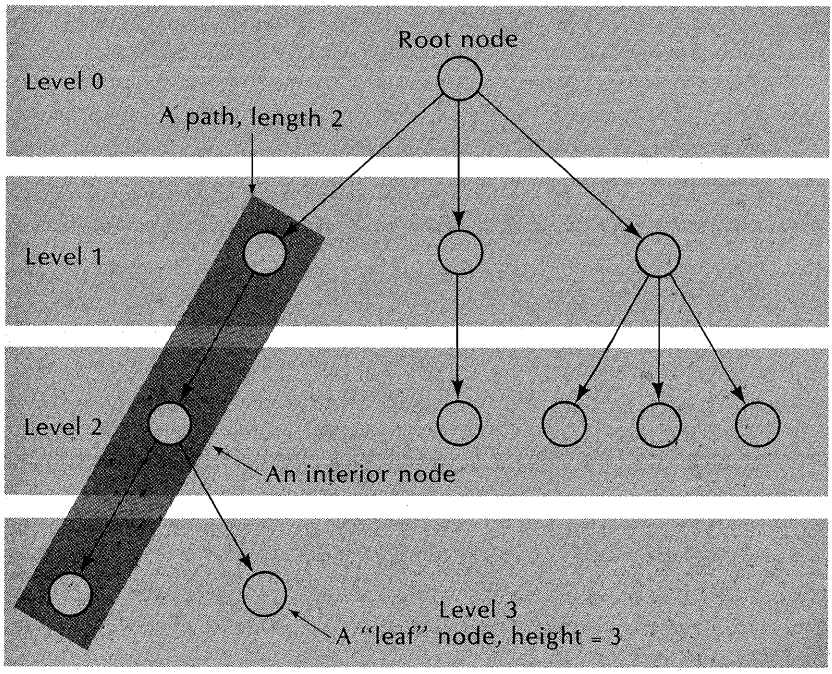


Figure 2.1. A typical tree.

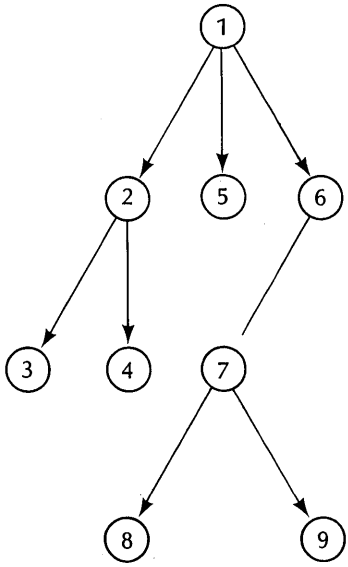


Figure 2.2. Preorder.

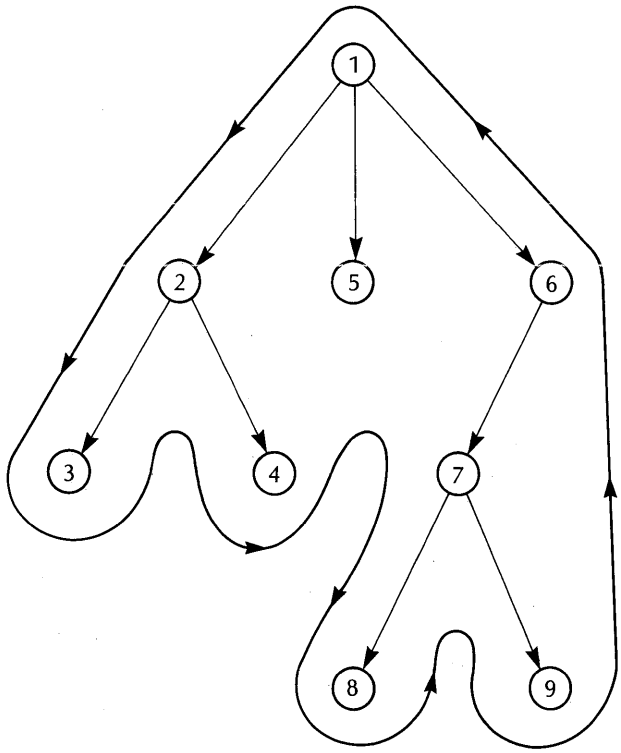


Figure 2.3. Obtaining preorder.

A tree node is said to be *decorated* when it carries some information in addition to its connectness. We may attach any sort of data to a node. In practice, we simply add more cells to each of the node elements shown in figure 2.4. For example, a cell may contain some simple data element or a pointer to some other data structure.

Exercises

1. The terminology of tree structures is obviously borrowed from certain properties of the plant phylum. If the root system of a plant must be included, is there a correspondence of the plant's components to a tree? Why not?

2. Consider the biblical injunction, "No man can serve two masters" (Matt. 6:24). If this applies to a business organization, would the

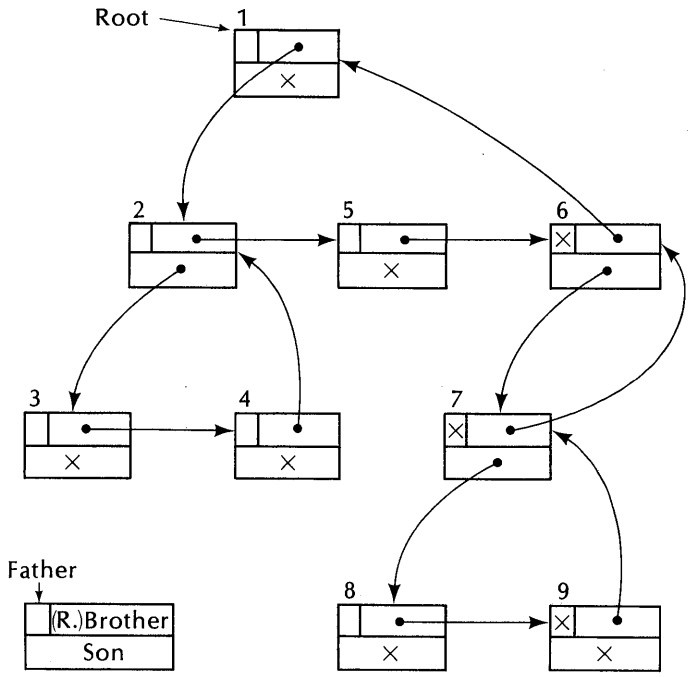


Figure 2.4. A simple pointer representation of the tree of figure 2.2.

organization correspond to a tree? What other conditions (if any) are needed?

3. Show that a tree must be acyclic, from its recursive definition.

4. Write a Pascal program that traverses a tree in preorder, given the pointer structure definition above.

Derivation Tree

A *derivation tree* displays the derivation of some sentential form in a grammar. Each node of a derivation is associated with a single terminal or nonterminal. A node associated with a terminal has no children. A node associated with a nonterminal may or may not have a set of children. Let N be a node associated with a nonterminal A, and suppose it has children. Then the children of N are associated with the symbols $x_1, x_2, \ldots, x_n$, where

$$A \rightarrow x_1\ x_2\ \ldots\ x_n$$

is a production in G.

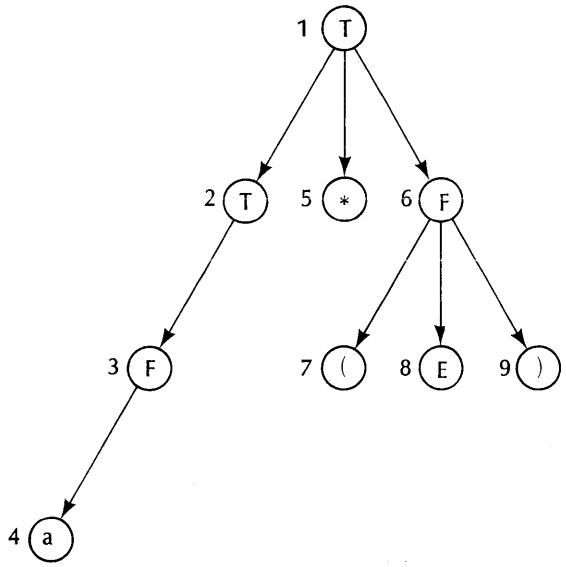


Figure 2.5. A derivation tree in grammar G_0.

Consider grammar G_0, given previously. A typical derivation tree in G_0 is shown in figure 2.5, rooted in the nonterminal element T. Its leaves, read in preorder, comprise the string

$$a*(E)$$

The string comprising the leaves of some derivation tree, in preorder, is called the *frontier* of the tree.

It is easy to show that the frontier of a derivation tree rooted in some token A is derivable from A in the tree's grammar.

We prove this by induction on the height of the tree.

Basis step. Let the height be 0. We then have $A \Rightarrow^* A$ in a derivation of zero steps, by definition of such a derivation.

Inductive step. Let the height be h, and consider some tree T of height $h+1$. Let N be the root of T. Since $h>0$, N must have at least one child. By the construction process of T, there is a production

$$A \rightarrow a_1\ a_2 \ldots a_n$$

where A is associated with N and a_i is associated with the i-th child of N. Now each of the i subtrees has a maximum height h, hence by the inductive hypothesis has a frontier f_i derivable from the token a_i. It should be clear that

the frontier of T is the left-to-right concatenation of the frontiers $f_1, f_2, \ldots, f_n$. But also

$$a_1\ a_2 \ldots a_n \Rightarrow^+ f_1\ f_2 \ldots f_n$$

Hence the frontier of T is derivable from the root token of T. QED.

The converse is also true. Given some derivation $A \Rightarrow^* w$ in a grammar G, we can always construct a derivation tree rooted in A with frontier w. The proof is left to the reader.

A picturesque way of looking at a derivation tree is to imagine that we have lots of *tree dominoes*, like the ones shown in figure 2.6. Each domino represents a production in the given grammar, and each part that carries a nonterminal is keyed so that it will only fit another domino with a matching key. The edges in each domino are made of rubber bands so that we may spread them apart as needed. We may start with any piece and build a tree downward from it. The terminal symbol parts cannot be connected to anything.

We assume that we have plenty of copies of each domino, so that we never run out of any one kind of domino.

A *complete* derivation tree for a grammar $G = (N, \Sigma, P, S)$ is such that its root node is associated with S and its frontier is a terminal string. The frontier is clearly a sentence in the language L(G). We shall normally assume that a derivation tree is complete, unless otherwise stated. A derivation tree may otherwise have a root node other than S, or its frontier may contain a nonterminal.

Exercises

1. Construct complete derivation trees for each of the following strings in $L(G_0)$:

   (a)+a
   a*(a+a)
   ((a))

2. Show that, given some derivation $A \Rightarrow^* w$ in a grammar G, there exists a derivation tree rooted in A whose frontier is w.

3. Can a derivation tree for a derivation in a context-sensitive grammar always be constructed? Why not? Give an example grammar and discuss.

4. Characterize informally the derivation tree of a right-linear grammar.

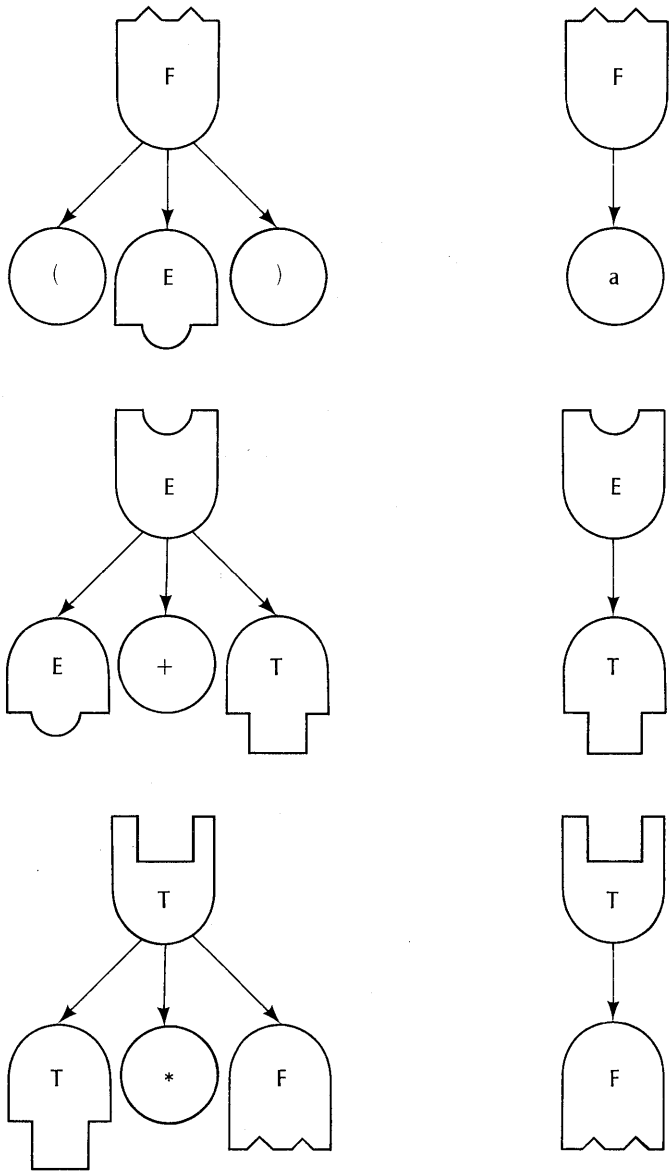


Figure 2.6. *Tree dominoes* for grammar G_0.

5. Figure 2.7 is a derivation tree for a sentence in some context-free grammar $G = \{N, \Sigma, P, S\}$ for which the productions and symbols are not known.

   (a) What is the frontier of the tree?

   (b) What symbols are necessarily in N?

   (c) What symbols are necessarily in Σ?

   (d) What productions are necessarily in P?

Syntax Trees

A *syntax* tree is a display of all of the productions in some grammar G. It contains two kinds of nodes, a *production* or P node and a *token* or T node. The root is a T node, and every path down from the root contains alternating P and T nodes.

A T node is associated with a terminal or nonterminal token A. It has children if A is nonterminal, and these are all the productions of the form $A \rightarrow w$. The children of a T node are P nodes.

A P node is associated with some production $A \rightarrow w$. Its children are the tokens in w, and these are T nodes.

Figure 2.8 shows a syntax tree for grammar G_0. T nodes are indicated by circles and P nodes by squares.

It should be clear that this defines an infinitely large tree; we can always add more nodes. However, we generally choose to consider a finite syntax tree in which each production appears exactly once. We build such a tree by starting with the root T node associated with the start symbol. We then add a production to the tree somewhere only if it does not already exist in the tree. The production consists of a P node and its T-node children. The tree's frontier then consists of T nodes.

An abbreviated represention for a syntax tree is shown in figure 2.9. Here, the P nodes have become vertical lines with horizontal branches, and the T nodes are simply the tokens of the production right part. This structure lends itself to a simple mechanical printing of a syntax tree from a set of productions.

A syntax tree may be transformed into a directed acyclic graph, called a *syntax graph* by adding directed edges as follows:

Given a nonterminal T node N with no children—it has no children because the productions normally connected to it appear elsewhere in the tree—add a directed edge from N to that T node associated with the same nonterminal symbol that does have a set of children.

Figure 2.10 shows G_0 represented as a syntax graph. We have simply added directed edges from nodes E, T, and F to their defining nodes in the tree of figure 2.9.

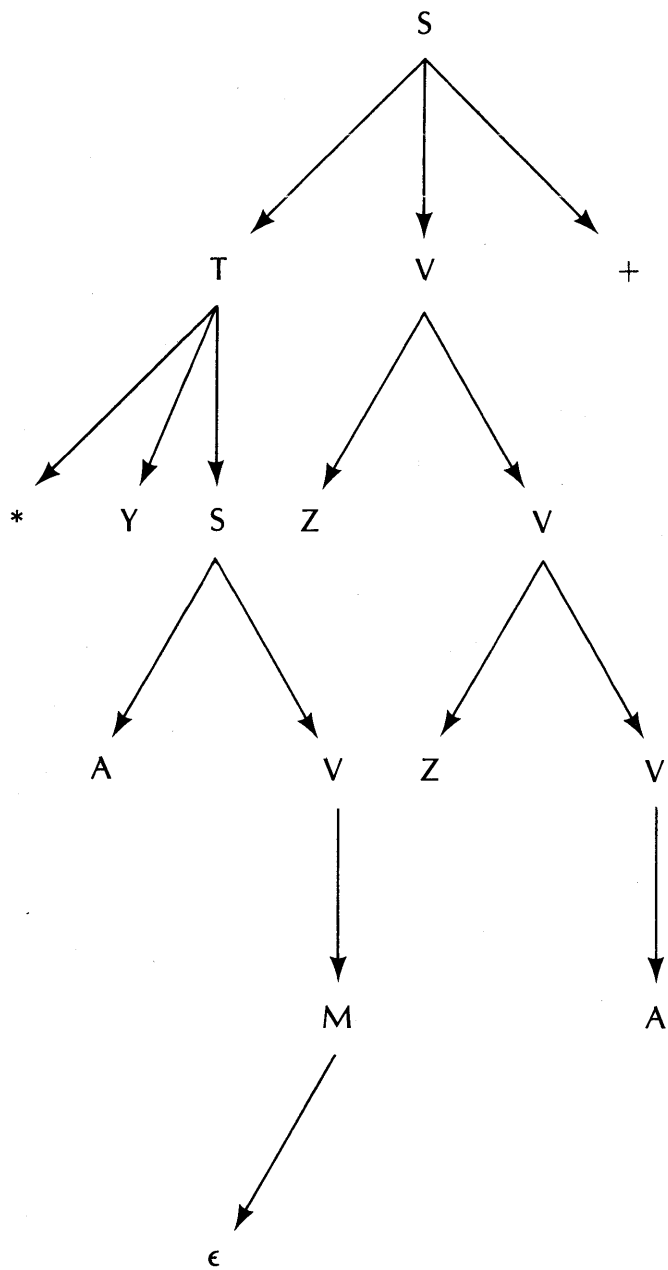


Figure 2.7. An example derivation tree.

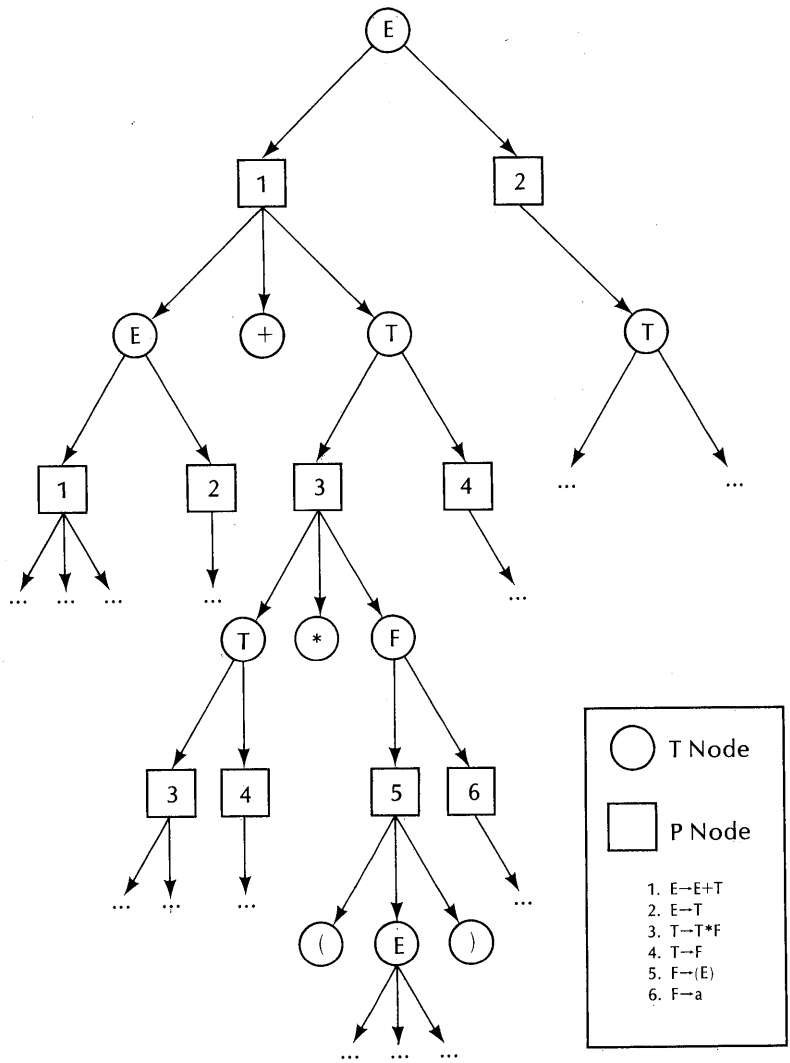


Figure 2.8. A production tree for grammar G_0.

A syntax graph is a useful and practical representation of a grammar. Cohen and Gotlieb (Cohen [1970]) show how sentences in a language may be generated or parsed by means of very simple procedures that interpret a syntax graph.

Semantic operations in a compiler must often be performed at just the right point during the sentence analysis. The operation is keyed by a particular production rule, and a syntax graph or tree enables us to easily determine the appropriate production rule for some operation.

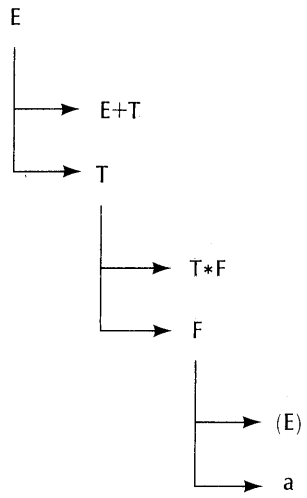


Figure 2.9. A finite production tree for grammar G_0.

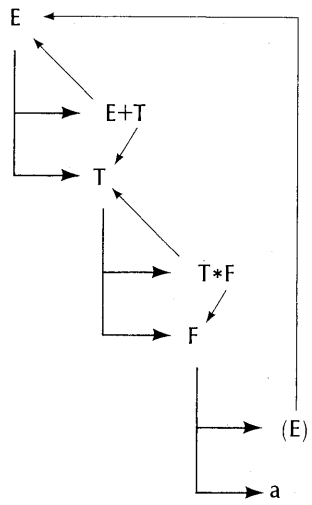


Figure 2.10. A syntax graph for grammar G_0.

Exercises

1. Consider a finite syntax tree constructed indefinitely far, but such that each T node has exactly one or zero children, and the leaves are T nodes. Show that its frontier is a sentential form.

2. Outline an algorithm that can accept a context-free grammar as input

and print its syntax tree in the form of figure 2.9. How might you deal with a finite page width and length?

3. A nonterminal X in a grammar G is said to be *useless* if and only if no terminal string can be derived from X. Develop an algorithm that identifies all useless nonterminals. (*Hint*—focus on the "useful" nonterminals; build a set of useful nonterminals; these either derive a terminal string or a string consisting of terminals and useful nonterminals in one step.) Give an algorithm for eliminating useless nonterminals from a grammar that preserves the grammar's language.

4. A nonterminal X is said to *inaccessible* if and only if no sentential form contains X. Develop an algorithm that identifies the inaccessible nonterminals and an algorithm for their elimination from the grammar.

5. Given a grammar G, possibly containing empty productions, find a transformation to an equivalent grammar G′ such that G′contains at most one empty production, $S' \rightarrow \epsilon$, where S′ is the start symbol of G′.

6. Given a grammar G, possibly containing single productions (of the form $A \rightarrow B$, where A and B are nonterminals), find a transformation to an equivalent grammar G′ such that G′ contains no single productions.

(Remark: The transformations of exercises 3 to 6 are important in reducing a grammar so that it is amenable to a precedence parsing method.)

2.2.6. Canonical Derivations

In general, a derivation step requires two kinds of choices to be made. We may have more than one nonterminal symbol in our sentential form, and for each nonterminal symbol, there usually is more than one production that may be used for the replacement.

For example, in grammar G_0, we can derive T*F as follows:

$$E \Rightarrow T \Rightarrow T*F$$

Now in the sentential form T*F, we may next replace either the T or the F. There is also more than one production with T as the left member (and with F as the left member).

The first kind of choice, that of which nonterminal to replace, has no effect on the class of sentence strings that can be derived from the start symbol. We may state this property as follows:

Given a sentential form xXyYz, where X and Y are in N, that can derive a sentence w through the derivation steps:

$$xXyYz \Rightarrow xryYz \Rightarrow^* w$$

then there exists a derivation

$$xXyYz \Rightarrow xXysz \Rightarrow^* w$$

and conversely.

Proof: In the derivation

$$xXyYz \Rightarrow xryYz \Rightarrow^* w$$

Y must be replaced somewhere. At that point we will have a sentential form

$$x'r'y'sz'$$

where $Y \Rightarrow s$, $x \Rightarrow^* x'$, $r \Rightarrow^* r'$, $y \Rightarrow^* y'$ and $z \Rightarrow^* z'$. We also know that

$$x'r'y'sz' \Rightarrow^* w$$

We may therefore reorganize the derivation as follows:

$$xXyYz \Rightarrow xXysz \Rightarrow xrysz \Rightarrow^* x'r'y'sz' \Rightarrow^* w$$

The converse is easily proven in a similar way. QED

This independence of the order of selection of nonterminals is a property of context-free grammars, but not of context-sensitive grammars. In a context-free grammar, each nonterminal can be expanded into some terminal string independently of its neighbors, and its expanded string essentially "pushes aside" its neighbors without interfering with their order in any way. Hence it doesn't matter which of several nonterminals in a sentential form are selected next for a derivation step.

We would like to have a standard derivation order, however, and each of the parsing methods to be introduced later has an inherent derivation order. Whenever we impose some ordering rule for the selection of the next nonterminal to replace in a sentential form, we have a *canonical* derivation. The two most common rules are *left-most* and *right-most*. In a left-most derivation, the left-most nonterminal in each sentential form is selected for the next replacement. In a right-most derivation, the right-most nonterminal is selected.

A top-down parse of some sentence, scanning from left-to-right through the sentence, corresponds to a left-most derivation. A bottom-up parse, scanning from left-to-right, corresponds to a right-most derivation in reverse order, i.e., the parser works from the sentence to the start symbol.

For example, in figure 2.11, we have effectively worked out the partial left-most derivation

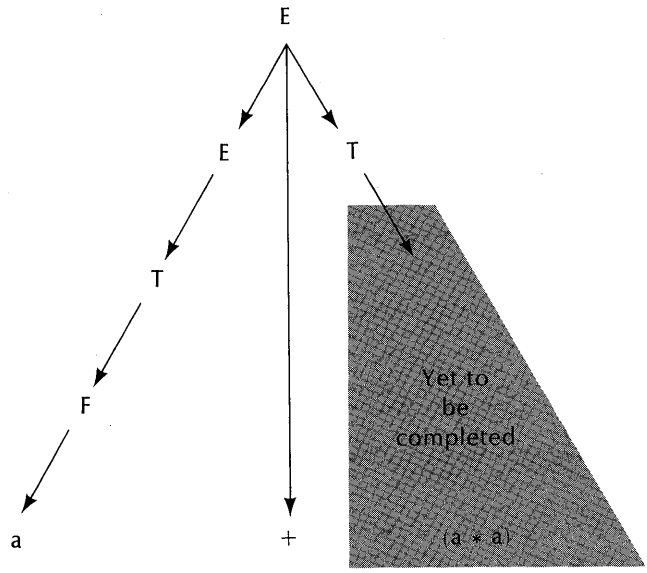


Figure 2.11. Top-down derivation tree construction.

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T$$

The remaining parse task is clearly $T \Rightarrow^{+} (a*a)$. The next derivation step must invoke the production $T \rightarrow F$, to yield

$a+T \Rightarrow a+F$

In figure 2.12, we have effectively worked out the partial right-most derivation

$$E+(F*a) \Rightarrow E+(a*a) \Rightarrow T+(a*a) \Rightarrow F+(a*a) \Rightarrow a+(a*a)$$

A bottom-up parser has developed this in backward order, starting with $a+(a*a)$ and ending (so far) with $E+(F*a)$. The next parse step should invoke production $T \rightarrow F$ on the right-most F, so that we will have the derivation step

$$E+(T*a) \Rightarrow E+(F*a)$$

Note that this is consistent with a right-most derivation.

Exercises

1. Give right-most and left-most derivations for each of the following strings in G_0:

   a*(E) {see figure 2.5}
   (a+a)*a
   a+((a))

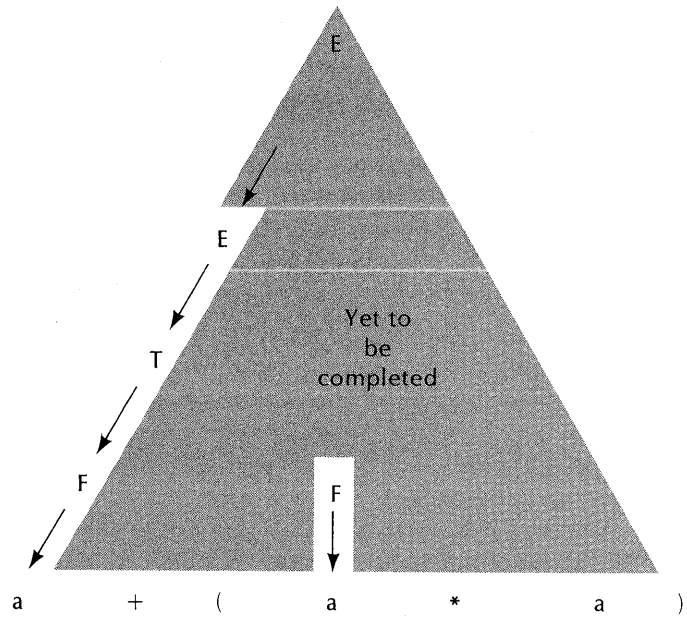


Figure 2.12. Bottom-up derivation tree construction.

2. Give yet another canonical derivation rule and illustrate a derivation in G_0 using your rule.

3. A left-most and a right-most derivation of some sentence w has the same derivation tree. Why?

2.2.7. Ambiguity

Suppose that we have a grammar and a sentence w for which two different derivation trees exist. By "different" we mean that the structure or the node labeling is different in some respect. We then say that the grammar is *ambiguous*. If no sentence has more than one derivation tree, we say that the grammar is *unambiguous*.

An ambiguous grammar is not a particularly desirable basis for a programming language. The meaning of a sentence lies mostly in its structure, as determined by the structure of its derivation tree, and not just in the set of symbols that comprise it. If there are two different derivation trees for some sentence, then it is possible that two different meanings can be attributed to the sentence.

English is full of ambiguous sentences, owing to the possibility of many words serving in different ways. For example,

"Time flies like an arrow."

can be interpreted in at least three ways: as an observation on the passage of time, as a command to compare the timing of flies with the timing of an arrow, or as a statement on the preferences of "time flies," whatever they are.

Similarly, a context free grammar may be ambiguous. For example, the grammar

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow a$$

derives exactly the same sentences as G_0, yet is an ambiguous grammar. Figure 2.13 shows two different derivation trees for the sentence

$$a+a*a$$

Now we can show that if two different derivation trees for some sentence exist, then there must also be two different canonical derivations for the sentence as well, and conversely. Thus in figure 2.13, we have the two different left-most derivations

$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$$

and $$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$$

We could similarly demonstrate two different right-most derivations.

We offer a left-most derivation proof of this assertion; a right-most proof is similar.

Theorem: Two or more distinct derivation trees for some sentence w exist if and only if two or more distinct left-most derivations exist for w.

Proof—"if" part. We have two distinct left-most derivations. The two agree exactly until some derivation step, in which the left-most nonterminal is replaced by one string in one and another string in the other, e.g.,

$$S \Rightarrow^+ uXv \Rightarrow uxv \Rightarrow^* w$$

or $$S \Rightarrow^+ uXv \Rightarrow ux'v \Rightarrow^* w$$

where x and x′ are different. Now consider the two derivation trees corresponding to these derivations. They may obviously be constructed top-down by following the derivation steps. The two trees are identical until the nonterminal node X is reached; its children are the string x in one tree and x′ in the other. Yet when the construction process is complete, both trees have the frontier w. QED

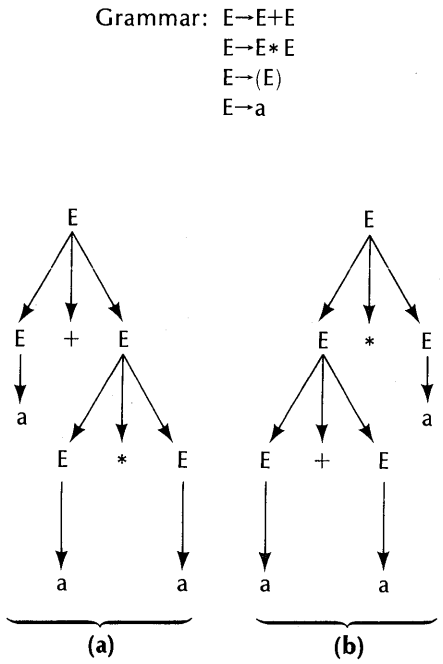


Figure 2.13. Two different derivation trees for the sentence a+a*a, through the ambiguous grammar given.

"Only if" part. Consider two different derivation trees T and T'with the same frontier w, rooted in S, and with the same grammar. We walk down through both trees (in preorder) starting at their root node, and continue as long as we find agreement, stopping on the first difference.

This tree walk is the same as the top-down construction process corresponding to a left-most derivation. If we are at some node N in T and it agrees with the corresponding node in T', then we consider the productions rooted in N and N'. If these fail to agree, we stop on node N. If they agree, then we compare each of the children, in preorder. We start the walk on the root node, and continue until the difference is found.

Now in the walk process, we have also generated two sequences of productions, corresponding to left-most derivations of the trees' frontiers. The sequences will agree until the tree difference is found, then there will be two different derivation steps, e.g.,

For tree T, we have

$$S \Rightarrow^* uXv \Rightarrow uxv \Rightarrow^* w$$

and for tree T', we have

$$S \Rightarrow^* uXv \Rightarrow ux'v \Rightarrow^* w$$

where the derivation $S \Rightarrow^* uXv$ corresponds to that portion of the tree walk just before the tree difference at node X is detected. QED

A language is said to be ambiguous if no unambiguous grammar exists for it. Note that a given language may have more than one grammar that describes it; some of these grammars may be ambiguous and others not. However, if one unambiguous grammar for a language can be found, then the language is unambiguous.

An important result in language theory states that there exists no algorithm that can accept an arbitrary context-free grammar and determine that it is either ambiguous or unambiguous. However, there exist algorithms that can return one of the results: {unambiguous, don't know}. These turn out to be parser constructor algorithms.

Exercises

These two exercises refer to the grammar $E \rightarrow E+E \mid E*E \mid a$

1. Three different derivation trees for the sentence a+a*a+a exist. Display them.

2. Suppose that ADD is emitted whenever production $E \rightarrow E+E$ is used in a left-most derivation, and MPY whenever $E \rightarrow E*E$ is used. "LOAD a" is emitted whenever $E \rightarrow a$ is used. Give the emitted code for the trees of figure 2.13, and discuss their significance.

2.3. Introduction to Parsing

A *parser* or *parsing automaton* is some system that is capable of constructing the derivation of any sentence in some language L(G) based on a grammar G. We are primarily interested only in parsers for right-linear and context-free grammars.

A parser may also be viewed as some mechanism for the construction of a derivation tree. However, we almost never actually construct a derivation tree in a practical compiler; instead the parsing algorithm makes use of a push-down stack and a finite state machine control.

Let us first look at parsing as a tree construction process.

2.3.1. Top-Down and Bottom-Up Parsing

The problem of structural analysis, or parsing, in a compiler may be seen as the problem of constructing a derivation tree, given a grammar and a sentence in the language. The sentence must form the frontier of the tree, and the tree will be rooted in the grammar's start symbol.

This is a nontrivial problem. Let us consider grammar G_0, whose productions are

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow (E)$$
$$F \rightarrow a$$

Then consider the sentence:

$$(a*(a+a))+a$$

Several different approaches may be taken. One might begin at the start symbol E and work downward toward the desired frontier. Many guesses are usually needed, and it will be found that a wrong guess somewhere in the process will usually result in an impossible situation. For example, figure 2.14 shows a partially constructed tree that appears reasonable, but in the end cannot possibly be right. Several mistakes were made in its construction. We still have to fit "a+a" into the inner parentheses, and have "+a" left over. We have found a partial tree for a sentence like (a*(a)), but it will not do for the sentence (a*(a+a))+a.

If we start at the bottom and work up, we also find ourselves making a number of guesses. It is certainly clear that every token "a" must be fitted to an F, since only one rule exists for that. It is also clear that somehow the left and right parentheses must be fitted into the production $F \rightarrow (E)$, since only that production contains parentheses. However, it is by no means clear (even with some practice) just how to fit these ideas together into a systematic plan for constructing a derivation tree.

In fact, a number of systematic derivation contruction methods have been discovered in recent years. We shall consider four of the most common and powerful of these in Chapters 4 and 5. These parsing methods fall into two broad classes—top-down and bottom-up. (A new parsing method, called *left-corner,* is an interesting blend of these two; see Rosenkrantz [1970a] and Demers [1976]).

Each of these methods reduces to the unit operation: "Determine a derivation step." Each may scan a sentence from right to left or from left to right. Now a sentence based in some grammar may be easily parsed from left to right, but with difficulty from right to left. It happens that most common programming languages are easily parsed from left to right, and furthermore, algebraic operations are usually performed in that order, by convention. We therefore confine our discussion to a left-to-right sentence scan.

Let us first consider the top-down, left-to-right parsing problem. A typical situation is shown in figure 2.11, for grammar G_0. We have already decided on the productions

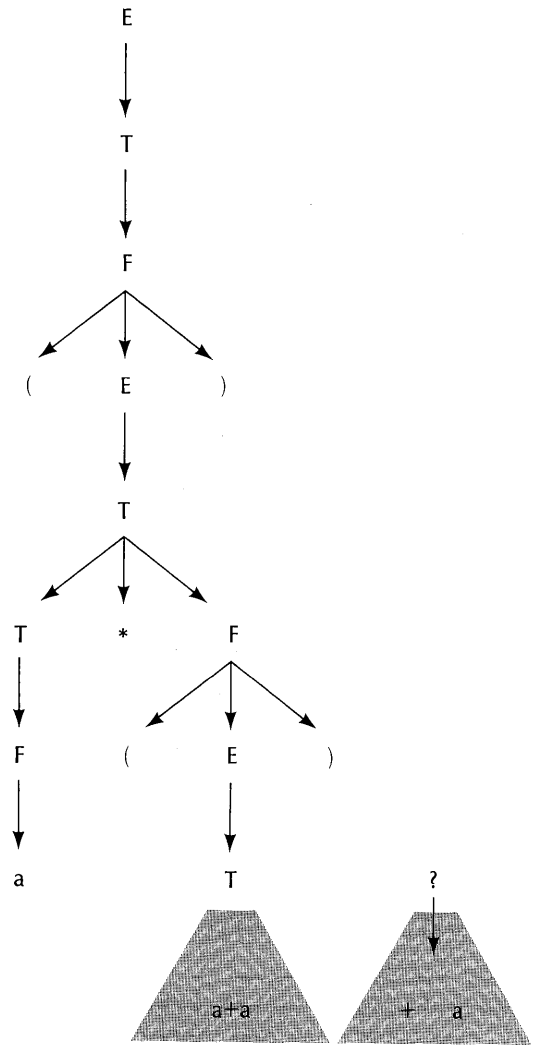


Figure 2.14. A bad guess for a derivation tree for sentence (a*(a+a))+a; top-down construction. The shaded parts cannot be incorporated in the tree.

$$E \rightarrow E+T$$
$$E \rightarrow T$$
$$T \rightarrow F$$
$$F \rightarrow a$$

and have accounted for the first two symbols a+ of the sentence a+(a*a). The left-most exposed nonterminal in the tree is T, therefore the parsing decision problem at this point may be stated:

Which of the productions $\{T \rightarrow F, T \rightarrow T*F\}$ should be connected to the exposed T node, given the partially constructed tree and the remaining input sentence (a*a)?

If we are somehow able to make the correct decision, given the information shown, each time, then we can repeat this operation again and again, until the tree is completely constructed. We would like to make each decision correctly by means of an algorithm, so that it will never be necessary to throw away our work and start over again. We also need some assurance, given any grammar, that we can find an algorithm and that it will work correctly for all the sentences in the grammar's language.

There are more considerations. What if it is possible to construct more than one derivation tree for some sentence? How can we be sure that the parsing method will reject sentences that are not in the language? These questions will be resolved when we study the top-down and bottom-up parsers in chapters 4 and 5.

Now let us consider the bottom-up parsing problem. Here we work from the given sentence upward toward the start symbol, in a left-to-right manner. We attempt to build a tree upward as far as we can before connecting productions to more sentence tokens. A partially constructed tree for grammar G_0 and the sentence "a+(a*a)" is shown in figure 2.12. We have decided that the productions

$$F \rightarrow a$$
$$T \rightarrow F$$
$$E \rightarrow T$$
$$F \rightarrow a$$

apply to the parsing process so far, and ask what the next production must be. It appears to be a more difficult decision than for a top-down parser—there are more possibilities. Should we go on to the next "a" token and apply another $F \rightarrow a$? Or should we extend the F tree through a production like $T \rightarrow F$? We can limit the choices somewhat by looking at the production right parts that can conceivably apply somewhere in the exposed tree, but in general, this still yields more choices than we can deal with.

However, we can still state the bottom-up parsing decision problem as follows:

Given a set of derivation trees (a tree may be an isolated terminal symbol or some tree rooted in a nonterminal), determine the production whose right member fits the left-most set of roots of the trees, and that "belongs" in the final derivation tree.

Neither the top-down nor the bottom-up parsing problem has a trivial or obvious solution. It is significant that these problems were not solved in a general way until about fifteen years after their statement.

2.3.2. Backtracking

The parsing problem can be seen as one of managing a sequence of choices in such a way as to find a set of choices that leads to a solution. For example, in figure 2.11, we have two choices of next production to be attached to the T node, T → T*F or T → F. Neither of these contains "(", "a" or ")". Although T → T*F contains "*", which we will need, it turns out that this choice is a poor one; the derivation tree cannot be finished if T → T*F is used at this point.

One approach to parsing is the general problem-solving method of *backtracking*. Let us first define the backtracking method in general, then show how it can be applied to top-down parsing.

Backtracking can be applied to any computation with these properties:

1. There exists a starting point and a goal.

2. The goal may be reached by starting at the starting point and following some path consisting of defined operations separated by nodes. At each node, some arbitrary choice among a finite set must be made. An operation leading from one node to another can either succeed or fail. We say the computation *blocks* if an operation fails.

3. Depending on the sequence of choices made, the computation will either reach its goal or block on some operation. If the computation blocks, we must back up one node and try another of the set of choices.

A backtracking machine will systematically explore all the choices and continue until either the goal is reached, or all the possible choices have been exhausted and lead to blocks. Unfortunately, the computation may continue forever. We need, in every application, some proof that the number of operations is bounded. We shall see that certain grammars will cause a backtracking parser to run forever on certain input strings.

There may also be more than one path to the goal. The path first found depends on the order in which the choices associated with the nodes are tried. An ambiguous grammar will yield a backtracking machine with multiple paths to the goal.

Let M be a generalized backtracking machine that contains a read/write tape used as a stack. Each cell of the tape will carry a *state* and a *choice*. Also, M manages the backtracking computation process by providing a systematic means of backing up and restarting when the process blocks.

The *process* will consist of a sequence of computations based on some algorithm, separated by choice points. At each choice point, a record is made on M's tape of the current state of the computation, and the particular choice made at that choice point. Each choice set must be finite and ordered in some

way so that it is always apparent, given a state and some choice, whether there is another untested choice.

The backtracking system then has these three moves:

1. A *forward* move from some state just recorded, using the particular choice selected by M. This will continue until: the machine blocks (step 2), it reaches its goal (halt), or it reaches another choice point (step 3).

2. A *backtrack* move, initiated by a block. Here, we consult the last-written M cell. If another choice exists in that state, we select it, record it, set the system to the state recorded in the cell, and do a forward move (step 1). If no more choices exist in that state, we remove the top cell from the tape. If the tape is empty, we halt (failure to reach goal). If the tape is not empty, we start again on step 2.

3. A *choice* move. Here we have reached a point at which some choice must be made. A new cell is added to the end of the tape containing the current system state and an *initial* choice (the first of the ordered finite set of choices available in this state). Then go to step 1.

We start the machine in step 3, by assuming that every backtracking process has an initial choice step.

In any machine application, we must show that the process will always halt in a bounded number of moves, otherwise we do not have an algorithm.

Application to Parsing a Context-Free Grammar

Let us apply our machine to the top-down parsing of a sentence in a context-free grammar. We have an input string $a_1 a_2 \ldots a_n$ of finite length. We also assume that we are building a derivation tree that will be accessible throughout the calculation. The tape will contain references to nodes in this tree.

At any point in the parse, the state of the system is the partial tree constructed so far, which incidentally includes the current position in the input string. We could conceivably just record the entire tree built so far on a tape cell, along with the particular choice of production made for the left-most exposed nonterminal node. However, such a move would be incredibly inefficient. We can accomplish the same result by storing only the two items: (1) current left-most exposed tree node, and (2) a production choice compatible with that node.

Step 1: The forward move. On a forward move, we have just chosen a production. We therefore attach it to the tree and examine the situation. Let the production be

$$A \rightarrow x_1 x_2 \ldots x_n$$

If x_1 is a nonterminal, we have reached another choice point, based on x_1, and therefore go to step 3. Otherwise, x_1 must match the left-most exposed input character. If it fails to match, we block and retreat to the backtrack move step 2. Each of the tokens $x_2, x_3, \ldots, x_n$ are similarly examined, until we either match all of them or we find the left-most nonterminal.

Suppose all of them match. We then search the partially constructed tree for a left-most exposed node. (This procedure will require an algorithm for moving up to the parent, seeing if it has any exposed right siblings, etc., the details of which will not concern us here). If a left-most exposed node exists, and is nonterminal, go to step 3 (a choice of production is needed).

If a left-most exposed node exists and is terminal, it must match the left-most exposed input token. On a failure to match, go to step 2 (backtrack). On a match success, continue matching.

Finally, suppose that no left-most exposed node exists. Now either the input string is completely attached to the tree or not. If it is attached, then we halt and report "success". Otherwise, we block and go to step 2.

Step 2: The backtrack move. On any backtrack move, we must discard a portion of the partially complete tree and try another production at the left-most exposed nonterminal node. We need the two items of information in a tape cell: the left-most exposed nonterminal node and the particular production chosen for it. Given the node, we can delete the subtree hanging from it and determine the input string position. Given the production choice, we can decide if another production choice exists. If it does, take it, record it, and go to step 1. If another choice does not exist, go to step 2 again.

Step 3: The choice move. The choice move is easy: we record the current left-most exposed nonterminal tree node, then select and record a production compatible with that node; it should be the first of an ordered set of compatible productions. (If the exposed node is associated with nonterminal A, then the compatible productions are all those with the form $A \rightarrow w$.)

Example. A grammar for which the backtracking system will work is the following (we shall later demonstrate two other ways to parse sentences derivable in this grammar). This grammar describes decimal numbers (containing a decimal point) with an optional sign.

Grammar $G_3 = (\{V, S, R, N\}, \{+, -, ., d, \perp\}, P, V)$, where P is

1. $V \rightarrow SR\perp$ {$\perp$ is a stop symbol}
2. $S \rightarrow +$

3. $S \rightarrow -$
4. $S \rightarrow \epsilon$ {ϵ is the empty string}
5. $R \rightarrow .dN$ {d is a decimal digit}
6. $R \rightarrow dN.N$
7. $N \rightarrow dN$
8. $N \rightarrow \epsilon$

Consider the input string:

$$+.dd\perp$$

A complete trace is shown in figure 2.15. There are no backups required for this string until the state shown in part (f) is reached. In trying the first choice ($N \rightarrow dN$) for the exposed node 9, we find that "d" and the next symbol $\perp$ fail to match, hence we must back up. The top cell says node 9 was given production 7 previously. Production 8 ($N \rightarrow \epsilon$) is still available, so we try it and find that we can match the remaining exposed node (the token $\perp$) and exhaust the input string.

Exercises

1. Trace the backtracking system on the strings

   $dd.d\perp$
   $-\ dd\perp$

2. Show that the backtracking system cannot accept either of the strings

   $d-\perp$
   $-\ +\perp$

3. Consider the following ambiguous grammar:

   $S \rightarrow E\ \perp$
   $E \rightarrow aEE \mid \epsilon$

   and its backtracking parser. Trace its behavior on the string

   $aa\perp$

   Discuss informally the factors influencing its choice of several possible parses.

Limitations of Backtracking

This system will succeed if and only if no left-recursive derivation in the basis grammar exists. A left-recursive derivation is such that

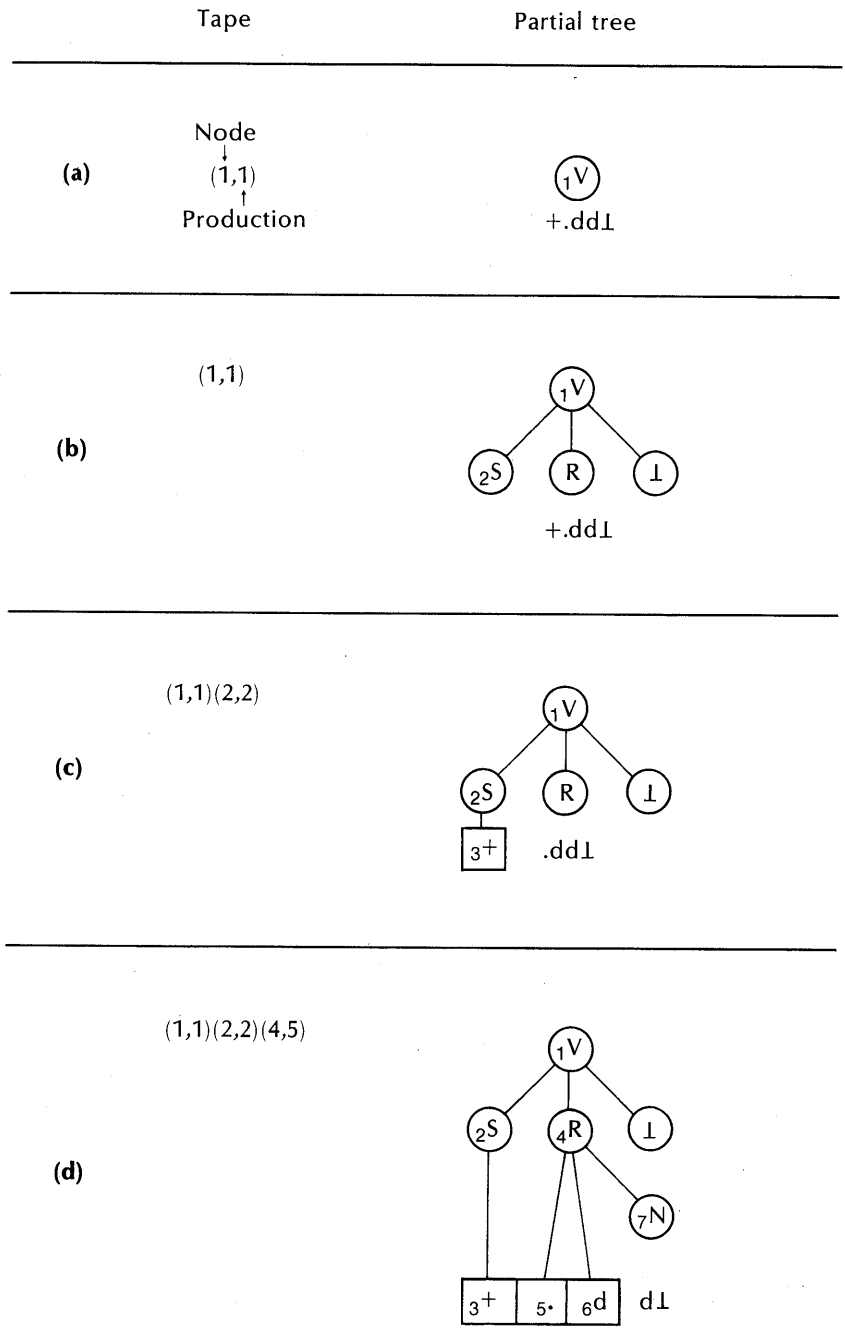
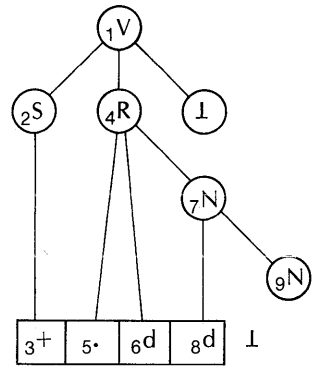


Figure 2.15. Trace of a top-down backtracking parse of a string "+.dd⊥" in grammar G_3.

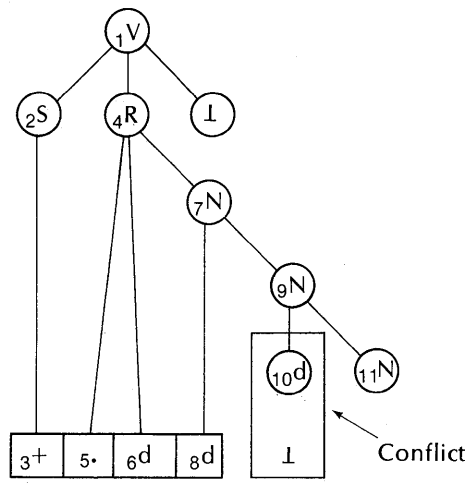
(e)

(1,1)(2,2)(4,5)(7,7)



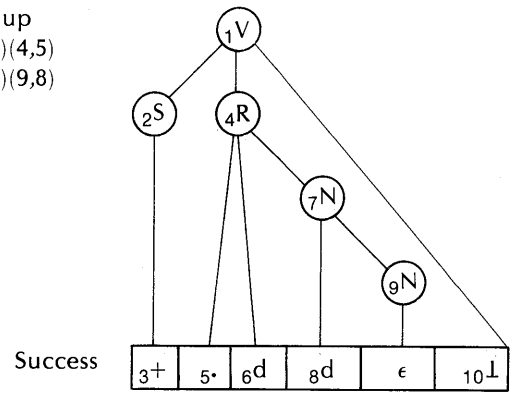
(f)

(1,1)(2,2)(4,5)
(7,7)(7,7)(9,7)



(g)

Back-up
(1,1)(2,2)(4,5)
(7,7)(7,7)(9,8)



$$A \Rightarrow^{+} Aw$$

for some nonterminal A. We shall not prove this; however, we can easily show that a left-recursion will cause a system failure. Consider grammar G_0 and the partial tree of figure 2.11. We have two productions compatible with the left-most exposed node T: $T \rightarrow F$ and $T \rightarrow T*F$. Note that the latter production is left-recursive. If we choose $T \rightarrow T*F$, we end up with another tree with the same exposed input string and the left-most exposed node T. The system will again choose $T \rightarrow T*F$ (assuming this is the first choice), etc. The tree and the backup tape will continue to grow indefinitely, with no progress in scanning the input string.

We might argue that the trouble lies in choosing $T \rightarrow T*F$ first. Why not arrange the productions so that left-recursive productions are chosen after the others? This choice seems to work in figure 2.11. We can add $T \rightarrow F$ to the tree and can continue nicely for awhile. In fact, by placing the left-recursive productions last among the choices, we can parse every string in the language $L(G_0)$.

However, this is not good enough. A parser must also be able to detect and report errors in syntax, i.e., it must be able to determine that some strings are not in the language. For example, suppose that in figure 2.11, the left-most exposed input string element were "*" instead of "(". We have a string that is obviously not in the language, and there is therefore no subtree that can be attached to the node T that will match a "*". What will the backtracking system do? It will attempt all possibilities. The $T \rightarrow F$ choice will eventually be found to fail (after many trials and errors), hence the choice $T \rightarrow T*F$ will be attempted. This choice, too, must fail because we eventually must get the same exposed input string and exposed nonterminal node as before and the system will run forever. We conclude that with a left-recursive production in the grammar, there are strings for which the system will never halt.

Aho [1972] shows that the backtracking system will never fail if the grammar is not left-recursive.

Time Bound

It can be shown (Aho [1972], chapter 4) that a parse of a string of length $n \geq 1$ for a non-left-recursive grammar will require no more than c^n operations. The c is some constant, > 1, that is characteristic of the grammar. This is a "best" bound, to the extent that we restrict the grammar in no way other than requiring that it be non-left-recursive.

Indeed, there exist grammars that cause the backtracking system to spend a

time proportional to an exponential of the length of the input. For example, consider

$$S \rightarrow cSS$$
$$S \rightarrow \epsilon$$

which derives sequences of c's. If Y(n) is the number of partial left parses consistent with string $w = cccc \ldots c$, where $|w| = n$, then Y(n) is certainly greater than 2^n (Aho [1972]).

This grammar causes the backtracking parser to construct and discard every possible partial tree before reaching its goal.

The behavior of any backtracking parser upon encountering a syntax error is also exponential in character. Since every possible partial tree consistent with the input string up to the position of the error must result in a block, the parser constructs and discards all of them. For a reasonably large grammar and strings of practical length, the time spent in such analysis is enormous.

We conclude that a backtracking parser system is impractical. We shall see that backtracking is unnecessary for a large class of grammars. There also exist more powerful generalized parsing methods (e.g., Earley [1968], also described in Aho [1972], chapter 4) that not only will parse any context-free grammar but do so with a better time bound than any backtracking system.

Exercises

1. Trace the backtracking parser for G_0 on the invalid string

   a**⊥

   far enough to show that it will never halt. Use the left-recursive productions last.

2. Trace the backtracking parser on the grammar

   $S \rightarrow cSS \mid \epsilon$

   for strings

   c
   cc
   ccc

   and discuss the parsing pattern it exhibits.

2.3.3. A Deterministic Top-Down Parser

The backtracking parser of section 2.3.1 is said to be *nondeterministic.* That is, given a choice at some node in the partially constructed tree, it simply makes an arbitrary choice and prepares for the possibility (the very likely possibility!) that its choice will be wrong.

Suppose that we had some way of making the correct choice each time. For a top-down parser, we have some information in the exposed input string that could be used to make the correct choice. For example, in figure 2.11, the exposed input string is (a*a), and this should be sufficient to determine that the correct choice of a T production is $T \rightarrow F$. We can then conceive of a large table such that each row is associated with a nonterminal node and each column with some legal input string. The table will then tell us which of several possible productions to choose for the next top-down move.

Unfortunately, such a table would be infinitely large–for interesting grammars, the number of possible unexposed input strings is infinite. For a practical compiler, we need a finite table.

Suppose instead we settle for a table such that every column contains only one input token, the left-most exposed string token, or *next token.* We clearly have potentially useful information in the rest of the exposed input string, but we can't use more than a finite amount of it anyway.

We still require that our table (now finite) fix a production choice for every possible situation. This requirement imposes a restriction on the basis grammar. It is possible to build such a table for some grammars and not for others.

Let us again consider grammar G_3, introduced in the previous section. Grammar $G_3 = (\{V, S, R, N\}, \{+, -, ., d, \perp\}, P, V)$, where P is:

1. $V \rightarrow SR\perp$ {$\perp$ is a stop symbol}
2. $S \rightarrow +$
3. $S \rightarrow -$
4. $S \rightarrow \epsilon$ {ϵ is the empty string}
5. $R \rightarrow .dN$ {d is a decimal digit}
6. $R \rightarrow dN.N$
7. $N \rightarrow dN$
8. $N \rightarrow \epsilon$

A top-down, one-symbol parsing table can be constructed for this grammar, by methods described in chapter 4. It is given in figure 2.16. Each row corresponds to a possible exposed left-most nonterminal node in the partially constructed tree. Each column corresponds to the next token. The entries are either a production number (1 through 8) or an X. The X means that there must be a syntax error; there is no way that a derivation based on the exposed nonterminal can match that token. For example, with token "."

Left-most Exposed Nonterminal	Next token: +	−	.	d	⊥
V	1	1	1	1	×
S	2	3	4	4	×
R	×	×	5	6	×
N	×	×	8	7	8

Figure 2.16. A top-down LL(1) parsing table for grammar G_3.

and nonterminal S, the table says that production 4 ($S \rightarrow \epsilon$) is the appropriate one to attach to the tree.

We can illustrate a parse without drawing a lot of trees. All we really need is the frontier of the partially constructed tree and the remainder of the input string. Thus figure 2.11 has the frontier a + T and the remaining string (a*a).

Given these two strings, we apply the table to the left-most nonterminal in the frontier and the first token of the remaining string, which yields a replacement string w. If the first tokens in w are terminal tokens, they must either match the tokens in the input string or else a syntax error exists. If they match, we drop the matched tokens before applying the table again.
Let us trace the process with the string −ddd.dd⊥.

Frontier	Remaining Input	Production
V	−ddd.dd⊥	1
SR⊥	−ddd.dd⊥	3
−R⊥	−ddd.dd⊥	(match, drop "−")
R⊥	ddd.dd⊥	6
dN.N⊥	ddd.dd⊥	(match)
N.N⊥	dd.dd⊥	7
dN.N⊥	dd.dd⊥	(match)
N.N⊥	d.dd⊥	7
dN.N⊥	d.dd⊥	(match)
N.N⊥	.dd⊥	8
.N⊥	.dd⊥	(match)
N⊥	dd⊥	7
dN⊥	dd⊥	(match)
N⊥	d⊥	7
dN⊥	d⊥	(match)
N⊥	⊥	8
⊥	⊥	(match and halt)

We stop and report "success" when both the tree frontier and the input list are empty. Other possibilities exist on input strings that are not in the language; for these the machine must report "failure." We require that every input string be terminated with the special symbol ⊥, and that this symbol not appear elsewhere in the input.

It can be shown that this parsing process has a time bound linear with the length of the input string, obviously a vast improvement over the backtracking approach of the previous section. However, we have paid for this time improvement with a certain restriction in the class of grammars that are amenable to this approach.

This parsing system is called an *LL(1) parser*, and was first described in Rosenkrantz [1970]. It has been used as the basis of several compilers, for example, see Lewis [1968]. "LL" means "Left-to-right, Left-most." The "1" refers to the single input symbol used to resolve the production choice. We could also use 2, 3, . . . symbols, yielding an LL(2), LL(3), . . . parser.

Exercise

1. Trace the parser of figure 2.16 on the strings

dd.d⊥	
.ddd⊥	
−.d⊥	
−.+d⊥	{illegal}
d.d+⊥	{illegal}
d.d.⊥	{illegal}

2.3.4. A Deterministic Bottom-up Parser

The bottom-up parsing problem seems more difficult than the top-down problem. There are more choices that must be made. Not only must we somehow select a production, but we must decide on the part of the partially completed tree that it applies to. Nevertheless, we can often construct a systematic deterministic bottom-up parser with no backtracking.

Let us begin with a definition: the *skyline* of a sequence of bottom-up parsing trees (see figure 2.12 for an example) is the left-to-right sequence of their roots. The skyline will always be a right-most sentential form, provided that the input string is a sentence in the language. For example, in figure 2.12, the skyline is E+(F*a), derivable from S by a right-most derivation.

Now we can introduce the bottom-up parsing machine, called an *LR(1)* parser. "LR" means "Left-to-right, Right-most"; the "1" refers to the

parser's need to examine at most one symbol in the input string past its current parsing point.

The parsing machine is shown in figure 2.17. It consists of a set of states (the circles) connected by transitions. Two kinds of transition appear: a *read* transition and a *lookahead* transition. The lookahead transitions are indicated by braces {...}. Note that the transitions are on the tokens in the terminal and nonterminal alphabets of the grammar G_3.

The states 1 through 8 (marked with a #) are called *apply* states. In an apply state, the associated production can be attached to the right-most exposed tree skyline. The states 9 through 17 call for a read or a lookahead transition.

We apply this machine to some partial tree skyline, starting with the *start* state 9 and continuing until we hit an *apply* state. On each read transition, we match the current skyline token against a transition token, then move to the next token and state. On each lookahead transition, we match the current skyline token against a transition token, and move to the next state, but do not move on in the skyline. A lookahead transition may carry more than one token. For example, in state 9, token "." or "d" is acceptable as a lookahead transition to state 4.

In an apply state, we have some production A → w indicated by the machine. The string w must then fit that part of the skyline we have just scanned. Notice that the machine "spells out" the string w upon falling into an apply state, associated with A → w.

As in the top-down parsing machine, we do not need to draw a lot of trees to illustrate the process. We need only show the skyline string. The initial

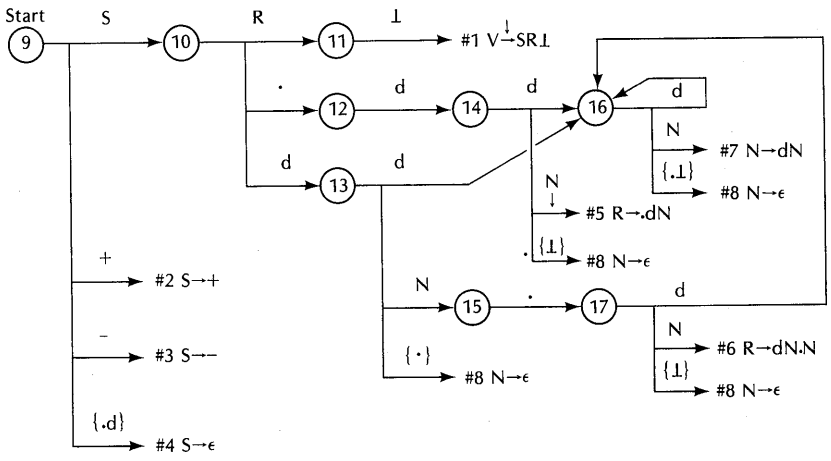


Figure 2.17. A bottom-up parsing machine for grammar G_3.

skyline string is the input sentence, and the final, or halt, state is upon applying production #1, $V \rightarrow SR\perp$ in the machine; the skyline will consist only of the start symbol V.

Let us trace this parsing machine with the input string ddd.dd⊥

Skyline	State Path	Production
ddd.dd⊥	9	4, $S \rightarrow \epsilon$
Sddd.dd⊥	9, 10, 13, 16, 16	8, $N \rightarrow \epsilon$
SdddN.dd⊥	9, 10, 13, 16, 16	7, $N \rightarrow dN$
SddN.dd⊥	9, 10, 13, 16	7, $N \rightarrow dN$
SdN.dd⊥	9, 10, 13, 15, 17, 16, 16	8, $N \rightarrow \epsilon$
SdN.ddN⊥	9, 10, 13, 15, 17, 16, 16	7, $N \rightarrow dN$
SdN.dN⊥	9, 10, 13, 15, 17, 16	7, $N \rightarrow dN$
SdN.N⊥	9, 10, 13, 15, 17	6, $R \rightarrow dN.N$
SR⊥	9, 10, 11	1, $V \rightarrow SR\perp$
V	(halt)	

It should be clear that we have reproduced a right-most derivation of the sentence, in reverse order.

A syntax error is detected in this machine whenever we cannot find a transition from some state that matches the next input symbol. For example, if we reach state 12 and fail to see a "d" symbol, then there must be a syntax error at that point, and the input string cannot be in the language $L(G_3)$.

We shall further discuss bottom-up parsing in chapter 5.

Exercise

1. Trace the parser of figure 2.17 on the strings

   dd.d⊥
   .ddd⊥
   −.d⊥
   −.+d⊥{illegal}
   d.d+⊥ {illegal}
   d.d.⊥ {illegal}

2.4. Bibliographical notes

Some early papers on grammars and generating systems are found in Chomsky [1956]. A survey paper with additional references is Chomsky [1963]. The notation used for grammars and derivations is from Chomsky [1959]. References for most of the remaining material in this chapter may be found in the notes for the subsequent chapters.

CHAPTER 3

FINITE STATE MACHINES

A large digital system cannot be designed through a detailed electrical analysis of all its circuits. There are just too many components and the electrical circuit laws are too difficult to solve. The system as a whole can only be understood by a model that simplifies the system. One such model is the finite-state machine. In this model, a digital system is viewed as one that moves in discrete steps from one *state* to another. Each transition is determined by the state it currently is in, along with a set of inputs. In the transition, the machine may also output some discrete set of values.

A state in a digital hardware system is defined by some finite set of signal voltages, interpreted in a discrete manner (usually *high* or *low*). A state in a software system might be defined by the set of values of the storage registers, including the current position in the stored program.

The finite state machine model has many applications. Every digital computer system is conceptually a finite state machine, albeit one with a vast number of states. Many seemingly difficult language recognition problems yield to a finite state machine synthesis. Many computer subsystems, such as peripheral device controllers, tape formatters, etc., are first designed as finite state machines that are then transformed into their logic circuit equivalents.

We shall examine in detail only one class of finite state machines—the so-called *incompletely specified* no-output machines. These are particularly useful as language recognizers. We shall see that the class of finite state machines, or finite state automata, (FSA for short) is equivalent in recognition power to the class of regular grammars, and also to a special class of language generators called regular expressions. Many simple programming languages can be recognized by FSA.

An Example FSA

Before we formally define a finite-state machine, let us examine one that will serve as an example for the formal descriptions to come, figure 3.1. This machine recognizes a language consisting of the signed or unsigned decimal numbers.

An FSA consists of a set of states, transitions among the states, and an input string scanned by a read head. The read head starts at the left-most string token and moves to the right as the FSA moves from state to state.

The circles containing letters represent states. At any one time, the machine is in exactly one state. The state S is a *start state*. The machine is

placed in this state initially. The states B and H are called *halt* or *accepting* states, and are so indicated by the double circles.

The arrows connecting the states represent *state transitions*. Each one is labeled with a member of the alphabet of the language recognized by the machine. For this machine, the alphabet consists of four tokens:

$$\{d, +, -, .\}$$

where d represents one of the decimal digits 0, 1, 2, . . . , 9.

As each token in the input string is scanned, the machine moves from state to state, according to the tokens on the arrows. For example, if the first token is "+", then the first transition is from S to A. Then if the next token is a decimal point, ".", the second transition is from A to G.

The FSA continues with its transitions until it either finishes the string, i.e., scans all the tokens, or it encounters a token that has no transition associated with it. If it scans the string and ends in a halt state (B or H), the string is said to be *accepted*. On the other hand, if it scans the string, but fails to end in a halt state, or if it is unable to scan the string because of a failure to find a matching transition on some token, then the machine fails to accept the string, and is said to *block*.

The FSA of figure 3.1 is *incompletely specified*, i.e., some states have no transitions on some tokens. For instance, state A has no transitions on "+" or "−".

The FSA of figure 3.1 is designed to accept those strings in the form of a signed or unsigned decimal number and only those strings. For example, it will accept these strings:

−15.
75.38
+.002
000001
+34.76

but will not accept these strings:

−75+
+17−56
3..14
.000.1

Now consider the specific string "+34.76", which is accepted by the machine of figure 3.1. The transitions are

S to A on token "+"
A to B on token "3" (a digit d)

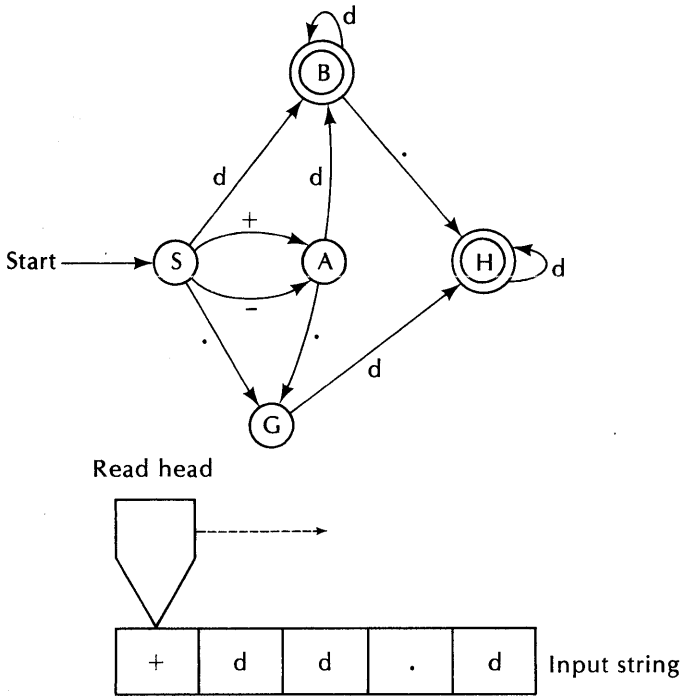


Figure 3.1. A finite-state automaton.

B to B on token "4"
B to H on token "."
H to H on token "7"
H to H on token "6"

Since H is a halt state, and we have succeeded in scanning the entire string, the machine accepts "+34.76".

Now consider the string "+17−56", which will not be accepted by the machine. The transitions are

S to A on token "+"
A to B on token "1"
B to B on token "7"

At this point the scan must end, since there is no transition from state B on token "−". There is only one transition on "−", from state S; however, it is of no value now because the machine is not in state S. State B is a halt state, but the input list must also be completely scanned, and it has not been. Hence the machine fails to recognize "+17−56"; it blocks on the "−".

Finally, consider the string "+." that will not be accepted. The transitions are

S to A on token "+"
A to G on token "."

The FSA has scanned the entire string, but has ended in state G, which is not a halt state. Hence the machine has failed to recognize the string "+.".

The value of such a machine in a computer system should be obvious—it provides a logically sound way to test input strings for membership in some language, that is, it serves as a *syntax checker.*

An FSA has many more applications. For example, we may associate an output string with each transition; we would then have a simple translator. We may also associate some general operation with each transition; such an FSA could then serve as a basis for a class of algorithms or as a machine controller, etc.

Exercises

1. Show that the FSA of figure 3.1 accepts these strings:

   75.38
   −15.
   00000001
   .000005

   but not these:

   +17−56
   −75+
   3. .14
   .00.1

2. Suppose that only state H is a halt state in the machine of figure 3.1. Describe the language of the FSA informally.

3. Suppose that only state B is a halt state. Using the definition of acceptance above, what is the significance of an input string that leads to state H? Is state H of any value to the FSA as a language recognizer? Show informally that it may be removed. What other state can also be removed? What language is recognized by the resulting machine?

4. Extend the machine of figure 3.1 to accept decimal numbers with an exponent field, e.g.,

$$+3.7E+6$$

3.1. Formal definitions

We now provide a more formal definition of a FSA, one that will be useful in exploring its properties. A *deterministic finite-state automaton*, or DFSA, is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states.
2. Σ is a finite set of permissible input tokens, i.e., the alphabet of the machine.
3. δ is a partial function that maps a state and an input token to another state, called the *state transition function*.
4. q_0 is a designated state in Q, called the *initial* or *start state* of the FSA,
5. F is a subset of Q, consisting of at least one final state.

The FSA operates through a sequence of moves. Each move is dictated by the present state and the next input token to be scanned by the machine. The move consists of scanning the next input token, and simultaneously transferring from a "current" state to a "next" state (which may be the same as the current state). A move may be made only if the δ function permits it to be made; the current state and the next token must map to another state through the δ function.

For example, consider the FSA of figure 3.1. Its state set $Q = \{S, A, B, G, H\}$, q_0 is S, its halt set $F = \{B, H\}$, its alphabet $\Sigma = \{+, -, d, .\}$, and its mapping function is:

$$\begin{aligned}
&\delta(S, +) = A\\
&\delta(S, -) = A\\
&\delta(S, .) = G\\
&\delta(S, d) = B\\
&\delta(A, d) = B\\
&\delta(A, .) = G\\
&\delta(B, d) = B\\
&\delta(B, .) = H\\
&\delta(G, d) = H\\
&\delta(H, d) = H
\end{aligned}$$

For example, the function δ maps state A and token "." to G. This corresponds to the transition A to G under "." in figure 3.1. The state transition function does not map all possible states and tokens to states; those state-token pairs that are not mapped are not permitted as automaton moves.

Transition Function as a Table

The transition function for a FSA may be expressed as a table. The input tokens are listed along the top and the states along the left side (figure 3.2). The table contains the mapping $\delta(P, a)$, where P (a state) defines a row and "a" (a token) defines a column. A blank entry means that δ is undefined for that state and input.

Configurations

Suppose that a FSA has completed a number of moves in a string. To predict its future behavior, we need only know the remainder of the input string, starting with the next token, and the current state. These two items of information provide a complete description of the FSA at a particular point in a particular application, and will be called a *configuration*. A configuration will be designated (q, w), where q is a state and w is the string remaining to be scanned.

The configuration (q_0, w), where q_0 is the start state and w is any string to be accepted or rejected by the automaton is called an *initial configuration*. A configuration (q, ϵ), where ϵ is the empty string, is called a *final configuration*, provided that q is in F, the set of halt states.

A *move* of the machine (designated "$\vdash$") connects one configuration to another. We have

$$(q, aw) \vdash (q', w) \text{ if and only if "a" is in } \Sigma, \text{ w is in } \Sigma^*, \text{ and } q' = \delta(q, a).$$

which means that given a machine in state q, with the input string "aw" ("a" is the first token and "w" is the rest of the string), one move results in state q′ and string "w". The token "a" has been scanned by the move, leaving the rest of the string "w". The move is only possible if the state transition function δ yields a state q′ for the current state q and input token "a".

A sequence of moves of the machine may be designated $\vdash^*$ or $\vdash^+$. The $+$ means "one or more moves," and the $*$ means "zero or more moves". A zero move results in no change in state and no scan of the input string. The sequence $\vdash^+$ is called the *transitive closure* of $\vdash$, and $\vdash^*$ is called the *reflexive transitive closure* of $\vdash$.

With this notation, we may succinctly define the language L(M) recognized by a FSA M:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q, \epsilon) \text{ for some q in F}\}$$

	δ	Input token +	−	.	d
	S	A	A	G	B
	A			G	B
States	(B)			H	B
	G				H
	(H)				H

Figure 3.2. Finite-state automaton of figure 3.1 as a table, expressing the transition function δ.

which means that the language L(M) is the set of strings w such that the FSA can begin in start state q_0, scan through string w, and end in a halt state when the string is completely scanned.

Exercise

Describe the acceptance of the string +.002 by the FSA of figure 3.1 as a sequence of configurations, starting with (S, +.002) and ending with (q, ϵ), where q is in {B, H}.

Machine Equivalence

Two machines, M and M′, are said to be syntactically equivalent if they recognize the same language, i.e., if L(M) = L(M′). The machines need not have the same number of states, nor must the states carry the same state labels. This definition is equivalent to the statement:

M and M′ are equivalent if and only if for every string x, M accepts x if and only if M′ accepts x.

If a machine M can be transformed into a machine M′ by merely relabeling its states, then M and M′ are said to be *isomorphic.*

A fundamental theorem of FSA is that for every machine M, there exists an equivalent machine M′ with a minimal number of states, and that every machine M″ with the same number of states as M′, and equivalent to M′, must be isomorphic to M′, i.e., M′ is structurally unique.

We shall expand upon this notion of equivalence later, and show how an arbitrary machine can be reduced to minimal form.

Nondeterministic Finite-State Automata

An FSA is said to be *deterministic* when no choices are provided in any of its moves. Every move is absolutely determined by the current state and the next token, clearly a desirable machine for any implementation. A *nondeterministic* FSA is such that some arbitrary choices are permitted in some of its transitions. There are some states and input tokens for which more than one transition may be taken. A number of concepts are easier to express nondeterministically than deterministically. We shall also show that, given a nondeterministic automaton, we can always systematically convert it into a deterministic automaton that recognizes the same language.

A *nondeterministic FSA*, or *NDFSA* for short, is defined exactly as a deterministic FSA, with two exceptions:

1. Some moves may involve a choice. This choice is represented by a state transition function δ that maps a state-next-token pair into a set of states. The set may consist of a single state, in which case no choice is provided for that particular state-next-token pair. However, in general some of the state-next-token pairs map to two or more states. We therefore use the notation:

$$\delta(q, a) = \{\text{some set of states}\}$$

for a nondeterministic transition function.

2. Some moves may be made without scanning the next token. Such a move is called an *empty move,* and may be included in the state transition function by the notation

$$\delta(q, \epsilon) = \{\text{some set of states}\}$$

There may be one or more possible next states. An empty move may be invoked (if it exists) even when the string has been completely scanned. In this way, it may be possible to reach a halt state through one or more empty moves from a non-halt state.

We say that a nondeterministic finite automaton accepts a string w if there exists some sequence of moves, beginning with the start state and ending in a halt state that scans the entire string. It is not necessary that each sequence of choices leads to acceptance; only one sequence is necessary.

Figure 3.3 gives an example of a nondeterministic FSA. Its transition table is figure 3.4. This machine happens to recognize the same language as the FSA of figure 3.1. We shall prove this in due time. Note that it contains a number of empty moves, S to A, A to E, etc. Also note that in state A, there are three possible moves on token "d". The machine may scan the "d" and transfer to either B or C, or it may make an empty move to E and then scan "d".

Now consider the recognition of the string "−24.57" by the NDFSA of figure 3.3. A correct choice would be the "−" transition from the start state S to A; the machine might also choose the empty transition to A. However, the

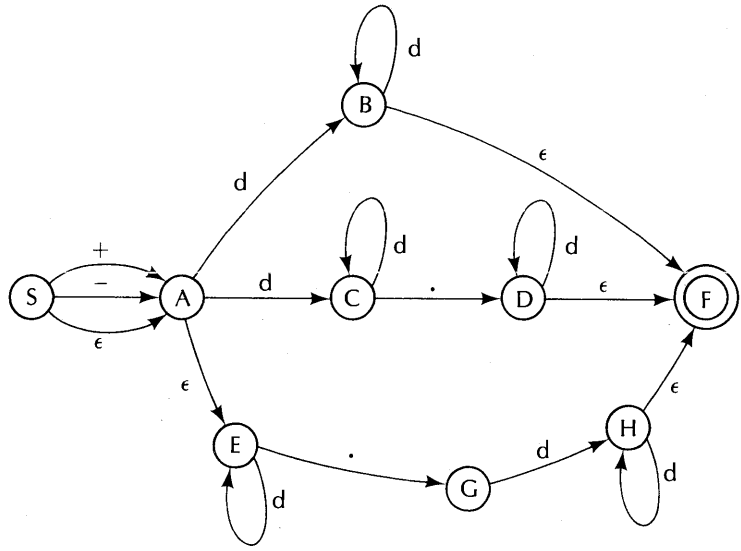


Figure 3.3. A non-deterministic machine equivalent to the machine in figure 3.1.

empty move results in a block, since there is no way to scan "−" once the machine reaches state A.

When in A, a choice among three possible paths exists, to state B, C, or E. Depending on the next token, any of the three may be possible moves. However, a move along the upper path, through B, means that a decimal point can never be scanned. An examination of the remaining two paths reveals that either one is satisfactory for our example, and results in acceptance of the string "−24.57". Thus, the middle path yields the state sequence A, C, C, D, D, D, F. Note again the nondeterminism of the final transition D to F. The machine is in D when the last token (a digit) is read, but may continue to make more empty moves to reach a final state (F).

Although it appears that a nondeterministic machine is more "loose" in its recognition capability, this is not really the case. We challenge the reader to find a string recognized by the automaton of figure 3.3 that is not recognized by the automaton of figure 3.1. The two automata are equivalent.

Exercises

1. Find a sequence of accepting moves for each of the following strings in the NDFSA of figure 3.3:

   +.004
   56.3
   −334

		Input Symbols				
	δ	+	−	.	d	ϵ
	S	A	A			A
	A				B,C	E
	B				B	F
	C			D	C	
States	D				D	F
	E			G	E	
	(F)					
	G				H	
	H				H	F

Figure 3.4. Tabular form of the non-deterministic finite automaton of figure 3.3.

2. Show that each of the following strings cannot be accepted by the NDFSA of figure 3.3, by exploring all the possible move sequences:

   +0.0.
   −2.+3
   ..02

3. Consider the NDFSA of figure 3.3, but with one of the three paths starting with A removed (there are three such machines). Discuss the three languages informally.

4. Design a simple NDFSA such that the acceptance of a string of finite length can be made in an indefinitely large number of moves.

A Backtracking Machine Model for a NDFSA

A nondeterministic FSA can be modeled by a backtracking system of the sort described in section 2.3.2. We do not propose implementing a FSA in this fashion; we merely present the model as another means of viewing a nondeterministic automaton.

Recall that a backtracking problem-solving system has three kinds of moves: a *forward* move, a *backtracking* move and a *choice* move. For a NDFSA, we may make each state transition a choice move, for the sake of generality, whether a state in fact has any choices or not. The backtracking move is invoked on any failure to accept the string. The forward move is simply a transition to the next state, scanning a string character in the process. The input string will be scanned left-to-right by a read head, however, the read head is permitted to move backward in a backtracking move.

The backtracking system tape T will have cells containing the state number, the position of the input token on the input list, and the particular state transition adopted, figure 3.5.

Initially, the tape T is empty and positioned at its left-most end; the machine M is in the start state; and the read head is positioned at the left-most token of the input list.

Upon leaving any state Q in a forward move, a cell on tape T is written.

If M blocks, the tape T is backed up cell by cell, until a cell is found such that an alternative move is found. The read head of M is then set to the position indicated by the cell, the state of M is set, the alternative move is made, and the cell is replaced by a new cell characteristic of the new move.

This process is repeated until one of two things happens:

1. The tape T is backed up to the first cell, and this cell indicates that no alternative moves exist. In this case, the input string is not in the machine's language.

2. The machine M ultimately reaches a halt state, and the input string has been completely scanned. In this case, the input string has been accepted by M. Tape T contains a record of the moves.

Now consider the NDFSA in figure 3.5, and let the transitions in each state be ordered from top to bottom in a clockwise sense. Thus transition "d" from A to B is labeled 1, transition d from A to C is labeled 2, and the empty transition from A to E is labeled 3. We also need string positions; let these be 1, 2, and 3, respectively:

–	3	.
1	2	3

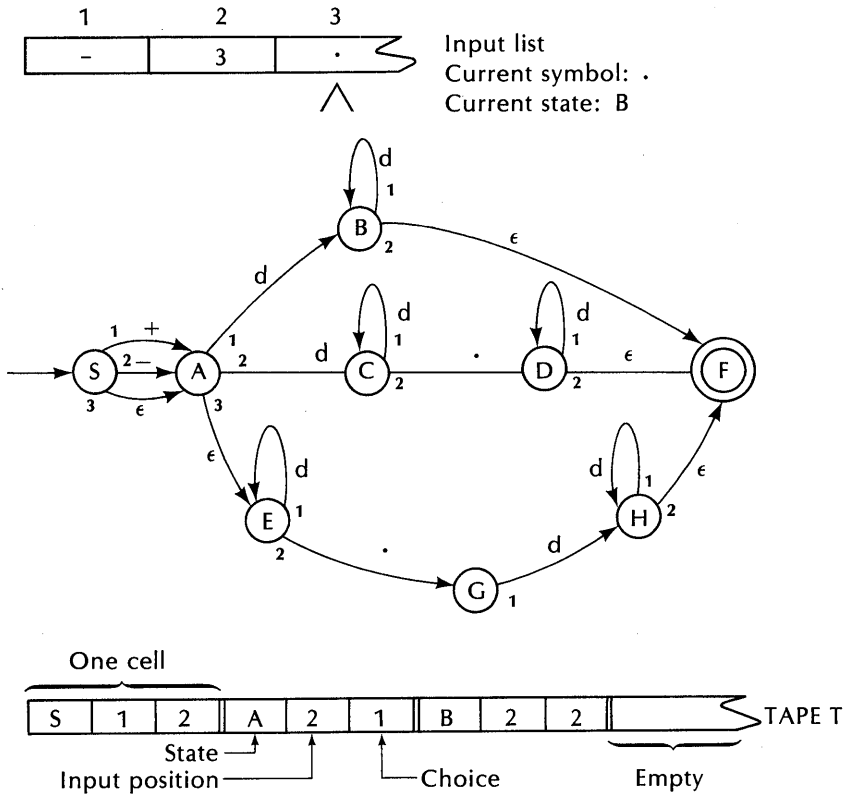


Figure 3.5. Backtracking machine M and its tape T, shown in one configuration. Input string: "−3.".

Let us trace some of the moves of the system for the above string. Initially, T is empty.

The first move from state S must be on the minus transition, 2, (first choice) so the first cell reads (S, 1, 2). The next move involves a digit "3" in position 2, from state A. We have three possible moves: to state B, to state C, or to state E. We take the first one, creating the cell (A, 2, 1). Tape T now reads (S, 1, 2) (A, 2, 1). The next move, from state B, calls for a decimal point. Although there is no such transition from B, the empty move to F may be taken. This yields the tape (S, 1, 2) (A, 2, 1) (B, 2, 2). Note that the input string position is unchanged. Figure 3.5 shows a snapshot of machine M, its input string, and the tape T at this point in the process.

We now find machine M in state F, with no more possible moves, and the input list incompletely read—which is not an acceptance condition, hence we must back up tape T and examine other alternatives. The last cell, (B,2,2), offers no hope, since there are no other moves out of B on the last token ".".

The next-to-last cell, (A,2,1), does provide another alternative. The cell indicates that the machine M chose the A to B transition on token "3". We can also move to C on this token, hence we do so, yielding the modified tape: (S, 1, 2) (A, 2, 2).

We now find the machine M in state C, with token ".". Clearly, the move from C to D (number 2 by the ordering scheme) is legal and yields the tape (S, 1, 2) (A, 2, 2) (C, 3, 2), with machine M in state D. The input list is now fully scanned; however, the empty move to F is legal, yielding acceptance and the final tape T = (S, 1, 2) (A, 2, 2) (C, 3, 2) (D, 3, 2).

As we shall see, it is never necessary to implement a nondeterministic finite automaton with a backtracking tape, because a nondeterministic finite automaton can always be transformed into an equivalent deterministic finite automaton. A deterministic automaton never needs to back up. If it blocks on a token, there exist no choices that have been made arbitrarily in its previous moves, and the block is therefore sufficient proof that the string is not in the machine's language.

Exercises

1. Trace the acceptance of the following strings through the backtracking system, figure 3.5:

   −.16
   3.6
   9.

2. Show that the following strings fail to be accepted by tracing the backtracking system:

   −.
   .3.

3. Show that the backtracking system will fail if an empty move cycle exists in the machine M, through an example. An *empty move cycle* is a sequence of transitions from some state A back to A, all with empty moves. Why will the system fail?

4. Show that the system will always terminate on a finite input string if no empty move cycle exists in the FSA.

3.2. Transformation of a NDFSA to a DFSA

The transformation of a NDFSA to a DFSA is accomplished by the following steps: (1) detection and removal of empty move cycles, (2) removal

of the remaining empty moves, and (3) transformation into a deterministic FSA.

3.2.1. Empty Cycle Detection and Removal

An *empty move cycle* is a sequence of empty transitions that begins with some state A and ends in state A. All the states in such a cycle are clearly equivalent, since we may get from any one of them to any other on any input token, without changing the read-head position.

An empty move cycle may be eliminated by merging their states. A set of states is *merged* by giving them all a common name. This has the effect of causing a transition into or out of any one of the empty cycle states to effectively be a transition associated with all of the states. If any one of the merged states is a halt state, the newly named state must also be a halt state.

Example

Figure 3.6 shows a machine with several empty cycles, ACD, etc. The ACD empty cycle may be collapsed by merging states A, C, and D. This merger yields figure 3.7, in which the ACED empty cycle has become the cycle AE. Collapsing this one yields the machine of figure 3.8, which contains no empty cycles.

Empty move cycles are detected and removed by the following algorithm.

Algorithm 3.1. Empty cycle removal

Let each state carry a mark in the set {0,1}. Mark 1 indicates that the state has been *considered*. Initially, every state carries mark 0 (*not considered*).

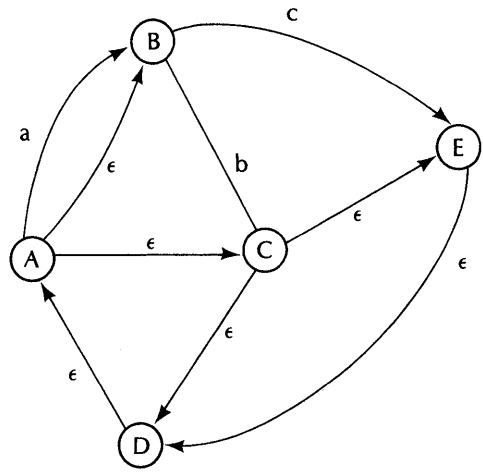


Figure 3.6. A finite-state automaton with several empty cycles.

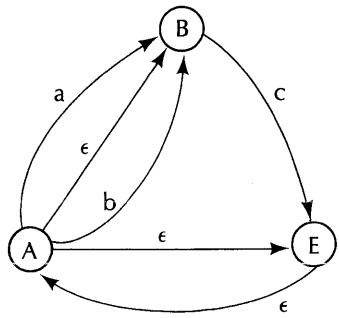


Figure 3.7. The finite-state automaton of figure 3.6 with the ACD empty cycle removed by merging states A, C, and D.

1. Choose any state p with mark 0. We then construct a tree whose nodes are states. Its root is p, and the children of any state q are those states for which an empty move from q exists. The construction of any path is terminated on a node with no empty moves, or on a node q′ such that q′ appears anywhere else in the tree (whether on that path or not). The tree is obviously finite, since it can contain at most as many nodes as there are states.

2. If node p appears twice in the tree, once as the root and again on some node N, then the path from the root to N represents the states on an empty cycle. Merge these states, and return to step 1. (Note that p remains unmarked).

3. If node p appears exactly once in the tree, as the root, then there are no empty cycles containing p. Mark p (1) and go to step 1.

The number of steps in this algorithm is clearly finite, and it is easy to show that at its conclusion, the machine M contains no empty cycles.

Exercise

Construct such a tree for the FSA of figure 3.6 and for each of the states B and C.

3.2.2. Removal of Empty Transitions

Once all the empty cycles have been eliminated, the remaining noncyclic

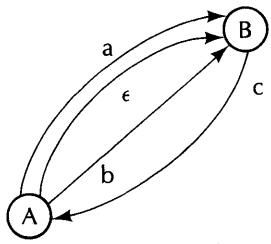


Figure 3.8. The finite-state automaton of figure 3.7 with all empty cycles removed by merging states.

empty moves may be removed. Consider some state p in machine M, with empty moves to states $q_1, q_2, q_3, \ldots$ (figure 3.9). This part of machine M is expressed by the transition function

$$\delta(p, \epsilon) = \{q_1, q_2, \ldots\}$$

Now each of the states $q_1, q_2, \ldots$ have transitions (in general) to other states $r_1, r_2, \ldots$ on tokens $a_1, a_2, \ldots$. Some of the r states may be q states or the p state, and some of the tokens $a_1, a_2, \ldots$ may be empty. However, no transition from p through q to itself can consist only of empty transitions.

Clearly, the empty move from p to q_1 can be eliminated if we add to p's moves the moves:

$$p \text{ to } r_1 \text{ on } a_1, \quad \text{and}$$

$$p \text{ to } r_2 \text{ on } a_2.$$

The idea is that if M can reach r_1 from p on an empty move and then on a_1, then an equivalent move is from p to r_1 on $a_1$directly.

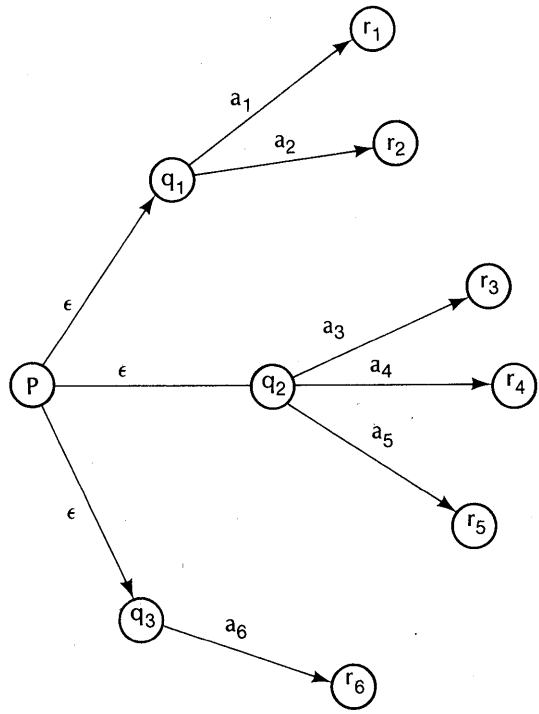


Figure 3.9. A state p with empty moves to states $q_1, q_2, \ldots$

Similarly, the empty move from p to q_2 may be eliminated by adding the moves

$$p \text{ to } r_3 \text{ on } a_3,$$

$$p \text{ to } r_4 \text{ on } a_4, \quad \text{and}$$

$$p \text{ to } r_5 \text{ on } a_5.$$

One more operation must be performed: If q is a halt state and an empty move from p to q exists, then p must be added to the set of halt states upon removing the empty move. We observe that if the input string is completely scanned when M is in state p, then M may move to q on no token and accept the string. If the empty move to q is removed, then this would no longer be possible; hence p must become a halt state.

The algorithm for this process follows.

Algorithm 3.2. Removal of empty transitions

Given an empty transition p to q on ϵ; i.e., q is a member of $\delta(p, \epsilon)$. Set $\delta(p, \epsilon) = \varnothing$, and add r to $\delta(p, a)$, for every a and r such that r is in $\delta(q, a)$. If q is in F, then p must be added to F.

This algorithm may result in one or more new empty transitions from p to some state r, and will therefore have to be repeated. However, it must ultimately end with no empty transitions from p, given that machine M contains no empty cycles. The argument is essentially that n repetitions of this algorithm involve state p, and a sequence of states r_1, r_2, etc., each of which must be distinct. If the r's were not distinct and different from p, then there must exist an empty cycle, a contradiction.

For example, consider figure 3.3. Although there are no empty cycles, there are five empty moves. Consider the empty move from H to F. There are no moves from F in this machine. However, since F is a halt state, H must become a halt state. Similarly, the empty moves from B to F and from D to F may be removed by making B and D halt states. We end up with a machine with four halt states B, D, H, and F. Since F can no longer be reached from the start state, state F is called an *inaccessible* state. There is no point in keeping inaccessible states, so machine M looks like figure 3.10 after removing these three empty moves and state F.

Next consider the empty move from S to A in figure 3.10. The rule is that we replace it with three new transitions from S to B, C, and E. Since A is not in the halt set, S is not added to the halt set. The result is the machine of figure 3.11, which still has two empty moves. One of them came from the empty move A to E. (Trust us—the reduction process is not caught in an infinite loop.)

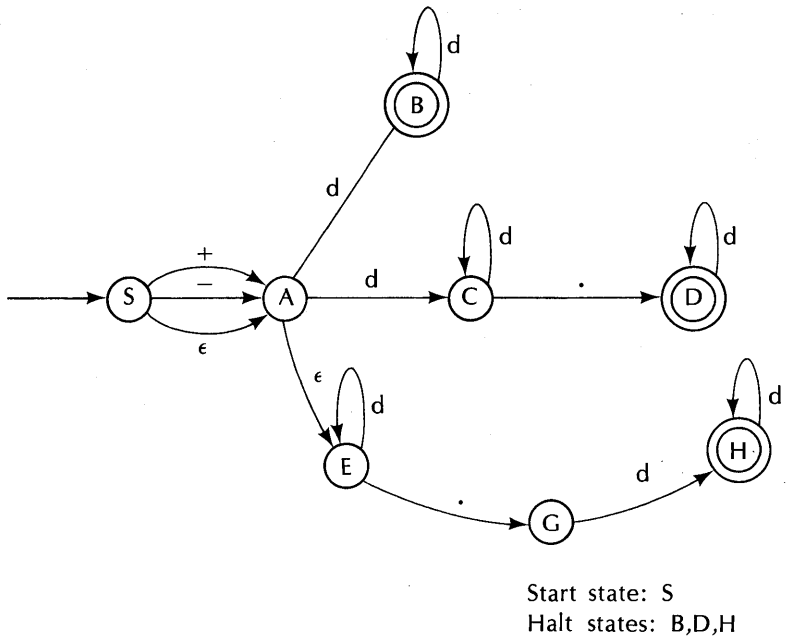


Figure 3.10. The finite-state automaton of figure 3.3 with the empty moves to F removed.

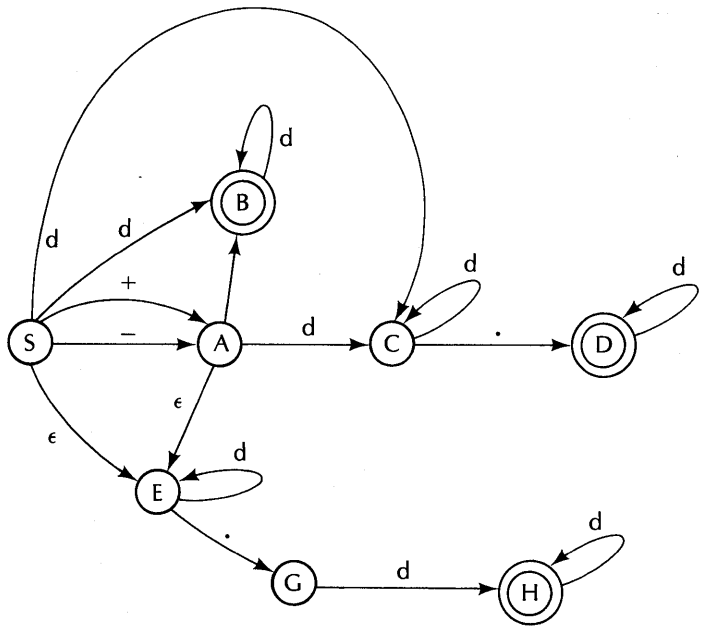


Figure 3.11. The finite-state automaton of figure 3.10 with the S to A empty moves removed.

Consider next the empty move from A to E. Its removal means that state A picks up transitions to E on "d" and to G on ".". Finally, the removal of the empty move from S to E means that state S picks up transitions to E on "d" and to G on ".". The final machine M′, free of empty moves, is shown in figure 3.12.

Removal of Empty Moves Using a Transition Table

A more systematic way of empty move reduction is through use of the table representation for the FSA. We first identify every state with an empty move to a halt state. When one is found, it is marked as a halt state. Considering figure 3.4, states B, D, and H have empty moves to F. Hence they can be circled, yielding figure 3.13. There are no other states that need be marked as halt states, since there are no empty moves to B, D, or H.

Now consider the empty move from state A to state E, figure 3.13. For token "+", δ(E, +) has no members. Hence δ(A, +) remains empty. Similarly for token "-". For token ".", δ(E, .) contains state G. Hence, we add state G to δ(A, .). In the same manner, for token "d", state E is added to δ(A, d). Figure 3.14 shows the resulting "A" row.

This operation is continued for every state with an outgoing empty transition, until no further additions to the table can be made. When this point is reached, all the empty moves may be dropped by crossing out the empty move column. Thus in figure 3.13, nothing is added to the table by the

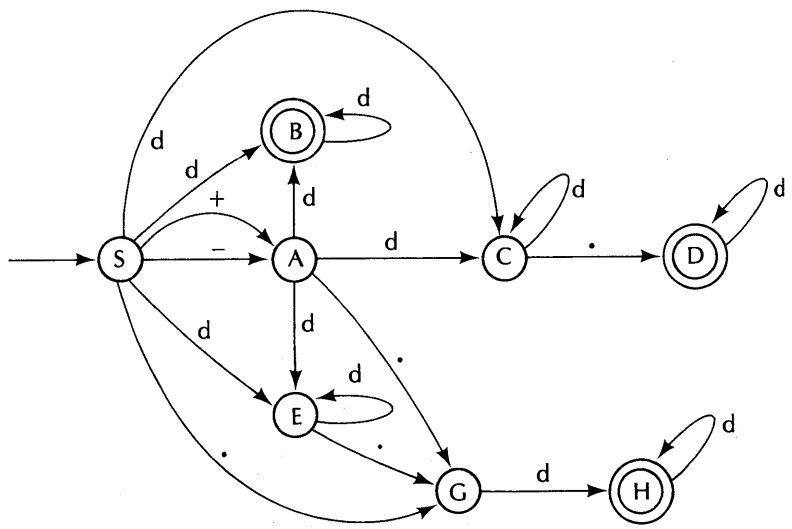


Figure 3.12. The finite-state automaton of figure 3.11 with all empty moves removed.

		Input symbols				
	δ	+	−	.	d	ϵ
	S	A	A			A
	A				B,C	E
	(B)				B	F
	C			D	C	
States	(D)				D	F
	E			G	E	
	(F)					
	G				H	
	(H)				H	F

Figure 3.13. Tabular form of the non-deterministic finite-state automaton of figure 3.5, with the halt states marked.

empty moves from B, D, and H to F, since F is empty anyway. However, the S row is expanded by the A row's states because of the empty transition from S to A. The final table, representing a FSA free of empty moves, is given in figure 3.14.

3.2.3. Transformation from Nondeterministic to Deterministic

The machine of figure 3.12 (or figure 3.14) is still nondeterministic. For example, there are three transitions from state A on a "d". Removal of the empty moves has not changed this situation; indeed, it has aggravated it.

The remaining nondeterministic moves of an NDFSA with no empty moves stems from one or more states with several moves on the same token possible. There are two such states in figure 3.12, S and A. We can resolve the choices in these states by calling each set of states a new state; the new state then will be the merger of its component states. Thus we create a new state "BCE" which will receive the merger of the transitions from states B, C, and E. We are in a sense deferring the choice on token "d" and state S by introducing a new target state "BCE".

The general method is defined in algorithm 3.3, as follows:

Algorithm 3.3. Converting a NDFSA M into an equivalent DFSA M′

1. The states of M′ consist of sets of states of M. That is, if A, C, and F are states in M, then {A}, {C}, {F}, {A, C}, {C, F}, {A, F}, {A, C, F} are states in M′. Since it is unusual to think of a set of states as a state, we change our notation slightly by using brackets [] instead of braces { } to represent a state in M′. Then [A,C] is the name of a state in M′, where A and C are states in M.

Although there are many possible sets of states of M, the maximum number is finite; indeed, if there are n states in M, then the largest possible number of states in M′ is $(2^n - 1)$. This can obviously be a very large number. Fortunately, most of the states in M′ are inaccessible and need never appear in the reduction process.

2. If P is a halt state in M, then every state [. . ., P, . . .] containing P in M′ is a halt state in M′.

3. If S is the start state in M, then [S] is the start state in M′.

4. Let $[P_1, P_2, \ldots, P_n]$ be a state in M′. Then consider all the transition functions

		Input symbols				
	δ	+	−	.	d	ϵ
	S	A	A	G	B,C,E	A
	A			G	B,C,E	E
	(B)				B	F
	C			D	C	
States	(D)				D	F
	E			G	E	
	(F)					
	G				H	
	(H)				H	F

Figure 3.14. Empty move removal. The ϵ column may be deleted.

$$\delta(P_1, a), \delta(P_2, a), \ldots, \delta(P_n,a)$$

on some token "a" in M. We then construct a new transition function δ' in M' as follows:

(a) Let $\delta(P_1, a) \cup \delta(P_2, a) \cup \ldots \cup \delta(P_n, a) = \{Q_1, Q_2, \ldots Q_r\}$. That is, we collect all of the states to which the states $P_1, P_2 \ldots, P_n$ transfer on token "a", and call these $Q_1, Q_2, \ldots, Q_r$.

(b) Then set $\delta'([P_1, P_2, \ldots, P_n], a) = [Q_1, Q_2, \ldots, Q_r]$. Note that this is a deterministic transition function, since the M' state $[P_1, P_2, \ldots, P_n]$ on token "a" transfers to exactly one state $[Q_1, Q_2, \ldots, Q_r]$.

5. Step (4) is repeated for every state in M' and every transition token "a".

This algorithm looks formidable, but in fact it is quite easy, particularly when carried out on a state table, as we shall see.

Tabular Reduction of a NDFSA to a DFSA

Consider figure 3.14. Its FSA may be transformed into a DFSA through row operations similar to those used for removal of the empty moves. The nondeterminism of figure 3.14 is accounted for by a multiple state set in the S and A rows, in the "d" column. According to algorithm 3.4, we need a new state [B, C, E] in M', since the set {B, C, E} appears in a transition in M. Let us therefore add such a state to the table. The transitions from the new state [B, C, E] consist:

- For token "+", of NULL, since neither of the states B, C, or E has a transition on token "+".
- For token "−", of NULL.
- For token ".", of a transition to state [D, G], since D is reached from C on token ".", and G is reached from E on token ".".
- For token "d", of a transition to state [B, C, E], since B goes to B on "d", C goes to C on "d", and E goes to E on "d".

Thus the new row for state [B, C, E] appears as shown in figure 3.15. Since the new state [B, C, E] contains a halt state (B) in M, it is marked as a halt state in M'.

In the process, we have introduced another state, [D, G]. This is a halt state. Its development in the table leads to the definition of another halt state, [D, H]. The algorithm ends on state [D, H], since every state appearing in the table is defined as some row in the table. The final DFSA is in figure 3.16.

Algorithm 3.3 generates a machine M' from a machine M, such that M and M' are equivalent. A proof of this assertion will be given later in the chapter, after we have defined equivalence more formally.

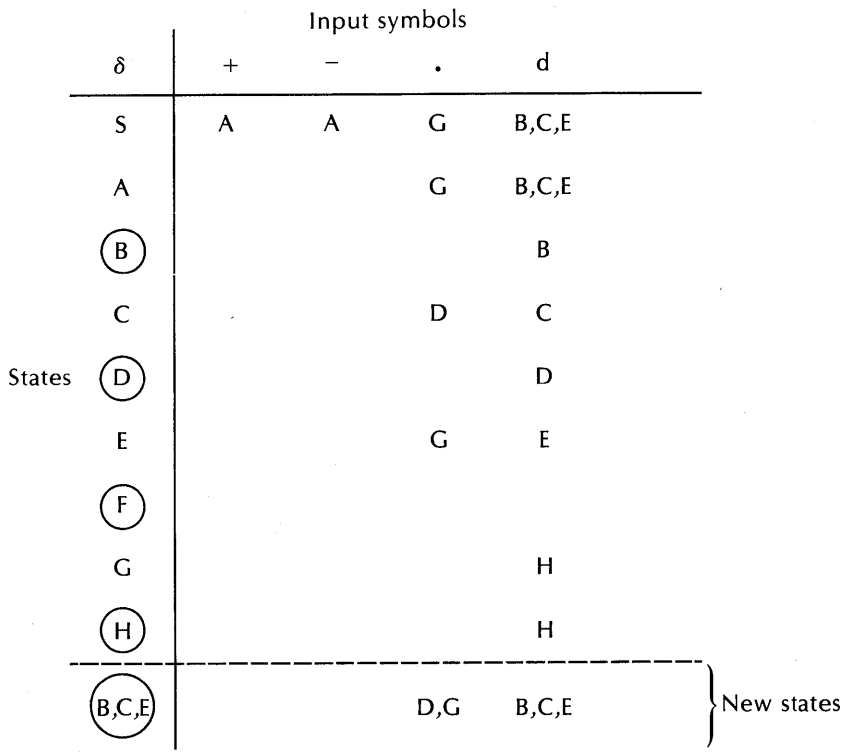


Figure 3.15. New composite state {B,C,E} created.

3.2.4. Accessible States

Some of the states in figure 3.16 cannot be reached from the start state, for example, state B. We may therefore delete all the inaccessible states from the DFSA, by using the following algorithm, which should be self-evident:

Algorithm 3.4 Detection of accessible states in a DFSA

1. Mark the start state S.
2. Given any marked state P, mark every state Q such that a transition from P to Q on some input token exists.
3. Repeat step (2) until no more states can be marked.

Upon completing algorithm 3.4, every nonmarked state is inaccessible from the start state, and may therefore be discarded.

Algorithm 3.4 applied to figure 3.16 shows that states B, C, D, E, and F are inaccessible. What happened to them? State F served only one purpose in the machine of figure 3.13–that of providing a halt state. But we have marked

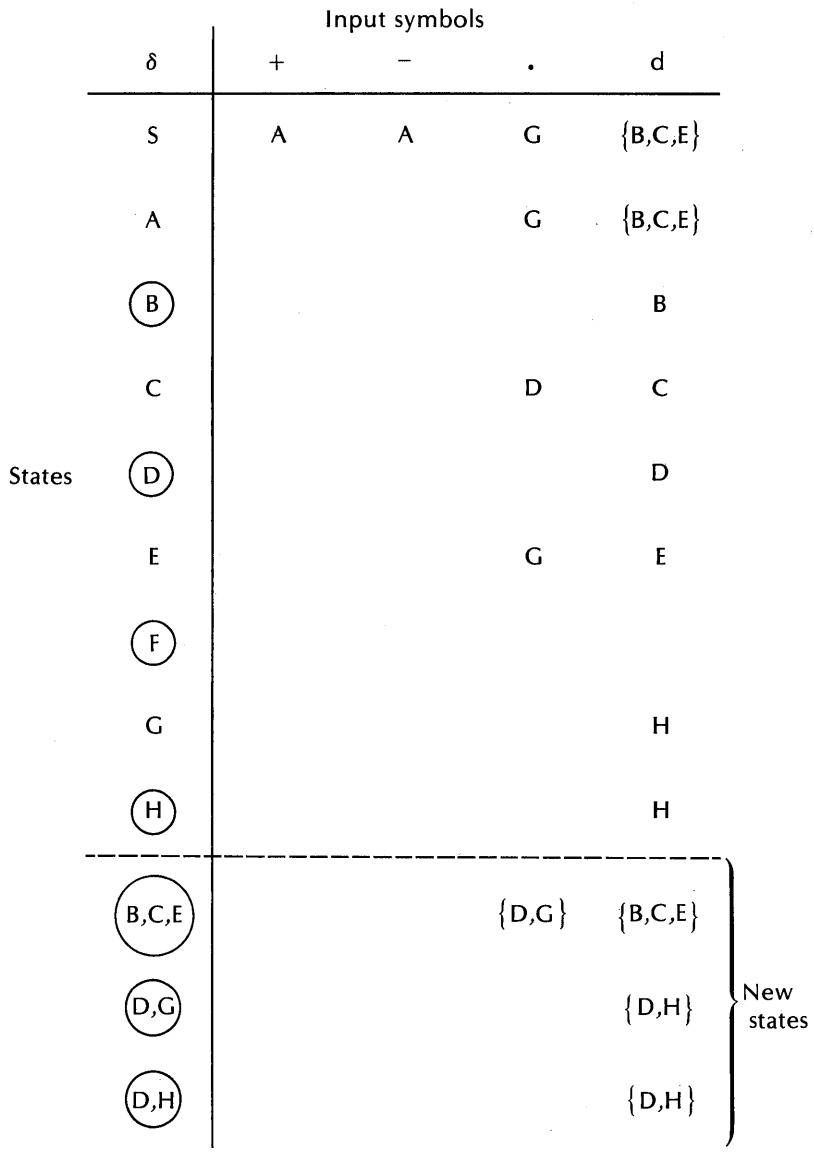


Figure 3.16. Completion of new state creation.

several accessible states in figure 3.16 as halt states, and they got that way through empty transitions to F. Hence F "lives on" in its presence in some other states.

State B, to examine another one, survives in the composite state [B, C, E],

similarly, C and E. State D survives in the composite states [D, G] and [D, H]. So while they are gone, they have left their mark on the FSA.

The DFSA, with its inaccessible states removed, is shown in figure 3.17.

Exercise

1. Reduce the following NDFSA to a DFSA, and remove the inaccessible states:

	input token					
δ	+	(	)	*	.	ϵ
S		A		C		
A	D		D			B
B	A	B			C	E
C	D		F			A
D		E			B	
E	C		F		D	F
F*						B

(the * indicates a halt state).

States	δ	Input symbols: +	−	.	d
	S	A	A	G	B,C,E
	A			G	B,C,E
	G				H
	(H)				H
	(B,C,E)			D,G	B,C,E
	(D,G)				D,H
	(D,H)				D,H

Figure 3.17. Inaccessible states removed.

3.3. Machine Equivalence

We have until now used the notion of machine equivalence without much development. Two machines M and M′ are equivalent if they accept the same language. That definition has sufficed thus far. We now develop the concept of equivalence more formally, and will arrive at an an algorithm for reducing the number of states in a finite-state automaton to the least possible. The reduction method will also enable us to decide whether two seemingly different finite-state machines are in fact equivalent. The general notion of language acceptance is not practical, since we can seldom try all possible strings on both machines and test their acceptances.

The reduction of a machine to the fewest possible states is obviously of economic value. Among other things, it will reduce the task of designing semantic actions for the machine to a minimum.

3.3.1. Definitions

We start by defining the *k-equivalence* between two states P in M and P′ in M′, where k is some integer, $k \geq 0$ and M and M′ may be the same machine.

State P in M and state P′ in M′ are said to be *k-equivalent* if, for every string x of length k or less, machine M in state P accepts x if and only if machine M′ in state P′ accepts x.

If states P and P′ are not k-equivalent, then they are said to be *k-distinguishable*; there is then some string x of length k or less, such that either: (1) machine M in state P accepts x, but machine M′ in state P′ does not, or (2) machine M′ in state P′accepts x, but machine M in state P does not.

Two states P and P′ are said to be *equivalent* if they are k-equivalent for all k.

A machine M is said to be *reduced* if no state in its state set is inaccessible and no two distinct states are equivalent.

A pair of equivalent states P and Q in a machine M may be merged by changing the name "Q" to "P" everywhere, without affecting the language recognized by the machine.

Let $P \overset{k}{=} Q$ denote k-equivalence of states P and Q in a FSA M. Obviously $P \overset{k}{=} P$, and by the symmetry of the definition, if $P \overset{k}{=} Q$, then $Q \overset{k}{=} P$. The k-equivalence is an example of a relation, and any relation that satisfies these two properties is said to be *symmetric* and *reflexive*.

The k-equivalence relation is also *transitive*: If $P \overset{k}{=} Q$ and $Q \overset{k}{=} R$, then $P \overset{k}{=} R$. It is easy to show that k-equivalence is transitive. Let x be any string of length k or less that is accepted in state P. Then if $P \overset{k}{=} Q$, it is also accepted in state Q. If $Q \overset{k}{=} R$, it is also accepted by state R. Hence $P \overset{k}{=} R$.

Any relation that is symmetric, reflexive, and transitive is called an *equivalence relation*. A fundamental property of an equivalence relation, and

one that we shall exploit in reducing a FSA to its minimal form, is the following:

An equivalence relation R upon a finite set of objects S partitions S into disjoint subsets, such that any two members of any subset are equivalent to each other, and no two members of different subsets are equivalent to each other.

This assertion may be proven by first considering how a set S is divided into subsets by an equivalence relation. Let the members of S be $a_1, a_2, \ldots, a_n$. Then we create a sequence of subsets $S_1, S_2, \ldots$ of S as follows. Each of $S_1, S_2, \ldots$ is initially empty. S_1 is created by placing a_1 into it, then including a copy of every other member of S that is equivalent to a_1. Note that by symmetry and transitivity, these must be equivalent to each other. When S_1 is completed, there may or may not be some members of $S\text{-}S_1$ left over. Suppose there are some members of $S\text{-}S_1$; call these $b_1, b_2, \ldots, b_m$. We then start a new subset S_2 by placing b_1 in it, then adding all the members of S that are equivalent to b_1. The interesting question is whether there can be a state that belongs to both S_1 and S_2. Suppose there were; let it be called Q. Then by transitivity and reflexivity, Q must be equivalent to a_1, because it is in S_1, and also to b_1, because it is in S_2. But then by transitivity a_1 is equivalent to b_1. We are led to a contradiction, since b_1 was specifically one of the states left out of set S_1 when we first collected together the states equivalent to a_1. Hence set S_1 and S_2 must be disjoint. A similar argument applies to sets S_3, S_4, etc.

Now suppose that we have somehow partitioned the state set of a FSA by k-equivalence. What is the nature of the partition induced by $(k+1)$-equivalence? The answer is that a $(k+1)$-equivalence is a *refinement* of a k-equivalence. By refinement, we mean either that the partition of a $(k+1)$-equivalence is exactly the same as that of a k-equivalence or that some of the subsets in the k-equivalence have become further subdivided. The boundaries between the subsets of a partition are not changed by a refinement; rather, additional boundaries are introduced.

Refinement may be illustrated as follows. Suppose we have a set S of states A, B, . . . , J, in a machine M, and they are partitioned as follows:

$$S = \{A, D, I\}\ \{B, C\}\ \{E, F, G, H\}\ \{J\}$$

Note that each state appears exactly once and belongs to exactly one subset. A refinement of this partition might be the following example:

$$\{A\}\ \{D, I\}\ \{B, C\}\ \{E, F\}\ \{G, H\}\ \{J\}$$

When at least one subset of a partition is subdivided in a refinement, the refinement is called *proper*. The above example is a proper refinement.

The following partition is NOT a refinement of the partition S:

$$\{A, B, D\}\ \{I, C\}\ \{E, J\}\ \{G, H\}\ \{F\}$$

Although there are more subsets, states I and C have become members of a common subset, whereas in S they were in disjoint subsets.

We now prove our assertion: that a (k+1)-equivalence induces a refinement on the partition induced by a k-equivalence. We need only show that a pair of states P and Q that were disjoint in the k-equivalence partition S remain disjoint in the (k+1)-equivalence partition S′.

To prove this assertion, recall that P and Q are disjoint in S because there exists some string x of length k or less that distinguishes these two states. But this string is also of length (k+1) or less, consequently P and Q must be disjoint in S′.

3.3.2. Reduction

We are at last in a position to reduce a FSA to its minimal form in a systematic manner. We need only these observations, which should be evident from the preceding discussion:

1. The 0-equivalent partition of the state set of a machine M is {F}, {Q-F}. That is, the 0-equivalent partition consists of the halt states and the non-halt states. These must be in separate partitions because the machine is in either an accepting or a rejecting state, depending on which state it is in, for a string of length 0.

2. As a partition is refined by identifying distinguishable states, eventually there must be a (k+1)-equivalence partition which is exactly the same as the k-equivalence partition. This reasoning follows because the state set is finite, and there are therefore a limited number of times a boundary can be introduced into a partition. For this k, the k-equivalent states must be equivalent (with no string length restriction), since no further refinement is possible. This partition is (k+1)-equivalent, (k+2)-equivalent, etc. Therefore each subset of this partition is a set of equivalent states.

3. A partition is refined by noting whether two states in the same subset can be distinguished by some single input token. For example, suppose P and Q belong to the same subset in a k-equivalent partition, and we find (by examining the state transitions), that P goes to P′ and Q goes to Q′ on some token "b". If P′ and Q′ are in different partitions, then they are distinguishable; consequently P and Q must be distinguishable and belong in different subsets in the (k+1)-equivalent partition. The two states may also be distinguished if one of them possesses a transition on some token, while the other does not. A nonexistent transition on a state P and token "b" means that the machine cannot scan string "b" in state P. If state Q has a transition on token "b" while P does not, then P and Q are distinguishable and belong in different subsets of the partition.

Summarizing, we begin with the two-fold partition of halt and non-halt states. Then we induce refinements on these by looking for single tokens that can distinguish two members of a subset. When no further refinements can be made, the machine has been reduced to minimal form.

Example. Consider the DFSA whose state transition table is given in figure 3.18.

The initial partition, on halt and nonhalt states, is

$$\{A, B, C, D\}\ \{E, F\}$$

We now attempt to refine this partition by looking for tokens that can distinguish pairs of states within either of the subsets.

Consider the pair (A,B). State B has a transition (to C) on "a", but A does not; hence these states belong in different subsets.

Next consider the pair (A,C). Again, C has a transition on "a", but A does not; hence, these belong in different subsets. Note that these conclusions do not prove that B and C belong in different subsets; hence, we must also consider pair (B, C).

States B and C have transitions on each of the three tokens. On token "0", they both transfer to E, on token "1", they both transfer to D, hence neither of these tokens serves to distinguish them. On token "a", state B goes to C and state C goes to B. Since B and C are in the same partition, token "a" also fails to distinguish them. We conclude that B and C belong in a common subset in the next partition.

Next consider states A and D, both clearly distinguishable. Also, since D has only one transition (on token "0"), it is distinguishable from B and C. Hence D belongs in its own partition.

δ	0	1	a
A	B	C	
B	E	D	C
C	E	D	B
D	F		
(E)		D	
(F)			

Figure 3.18. A machine to be reduced.

Finally, consider states E and F. State E has a transition on "1" not possessed by F, hence this pair is distinguishable.

Our 1-equivalent partition therefore looks like this:

{A} {B, C} {D} {E} {F}

There is only one subset that is a potential candidate for a partition, the {B, C} pair. A glance at the table shows that this partition cannot be refined. Hence these two states must be equivalent. The reduced machine has five states. State C may be renamed "B" wherever it appears. The reduced machine is in figure 3.19.

Another Example. Consider the decimal number machine in figure 3.17. Its initial partition is

{S, A, G} {H, BCE, DG, DH}

States S, A, and G are clearly distinguishable. Similarly, state BCE belongs in its own partition. However, what about H, DG, and DH? They all transfer to a common subset on token "d", whether in this partition or the next one. The final partition is then:

{S} {A} {G} {BCE} {H, DG, DH}

and the triplet subset cannot be further partitioned. Hence the final machine has five states, with two halt states, as shown in figure 3.20, which may be compared with figure 3.1; these two machines are clearly isomorphic.

δ	0	1	a
A	B	B	
B	E	D	B
D	F		
(E)		D	
(F)			

Figure 3.19. Machine of figure 3.18 reduced.

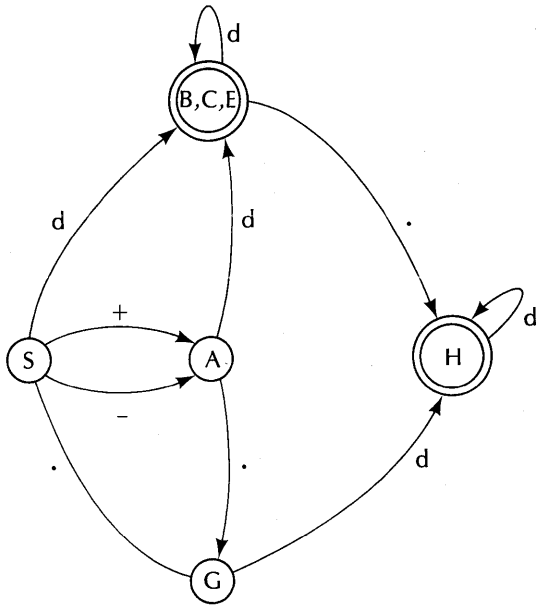


Figure 3.20. The machine of figure 3.12 made deterministic and reduced.

Summary

Recall that we asserted that the machine of figure 3.1 was equivalent to the nondeterministic machine of figure 3.3. We have now demonstrated that assertion, through the following machine transformation steps. These steps provide a systematic way to construct a program to recognize an important class of languages.

- Removal of empty move cycles (if any).
- Removal of empty moves.
- Removal of nondeterminism.
- Removal of inaccessible states.
- Reduction by identifying and merging equivalent states.

Since this process may be applied systematically to any FSA, and we have assurance that it will always yield a machine with the minimum number of states, it is possible to determine whether two different-appearing machines are in fact equivalent. We merely reduce each of them by the above process, then test them for isomorphism. We leave the matter of testing isomorphism for an exercise.

Exercises

1. Reduce the following FSA:

state	input 0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

2. Show that if two states P and Q are k-distinguishable for $k \geq 0$ then they are $(k+1)$-distinguishable.
3. A machine M has n states. Give a bound for the largest k such that some pair of states is $(k-1)$-distinguishable but k-equivalent.
4. Develop an algorithm that tests two reduced machines M and M′ for isomorphism.
5. Develop an algorithm that tests two nonreduced machines M and M′ for equivalence. *Note*: The obvious approach is to reduce each of them, then apply the solution to exercise 4. Is there a more direct approach?

3.3.3. A Systematic Reduction Method

Several systematic tabular methods for machine reduction exist. We describe one that can be programmed easily on any computer, called the *pairs table* method.

A *pairs table* contains a pair of states or a *null* at the intersection of each row and column. Each column is associated with an input token. Each row is associated with a *feasible state-pair*. The reduction consists of two algorithms, one that builds a pairs table for the nonreduced deterministic FSA and one that marks certain rows. At the conclusion of the second algorithm, each row that is not marked is associated with a pair of equivalent states; each row that is marked is associated with a pair of distinguishable (nonequivalent) states. Furthermore, all the equivalent states appear as unmarked state-pairs in the final table.

A *feasible state-pair* is a pair of states that conceivably could be equivalent upon a cursory examination of the FSA transition table. More precisely, a state-pair (p,q) is a member of the feasible state-pair set if (1) {p, q} is a subset of either F or Q-F, i.e., both are halt states or both are non-halt states, and (2) for every input token "a", $\delta(p, a)$ is $\varnothing$ if and only if $\delta(q, a)$ is $\varnothing$.

By selecting only those state-pairs that satisfy these two conditions, we eliminate from further consideration all those pairs of states that are obviously distinguishable, e.g. (1) a halt state is distinguishable from a non-halt state, and (2) a state with a transition on some symbol is distinguishable from another without a transition on that symbol.

For example, consider figure 3.17. The state-pair (S, A) is not a feasible state-pair, because S has a transition to A on token "+", while A does not. The state-pair (G, H) is not feasible because H is a halt state, while G is not.

Thus the set of feasible state-pairs for the machine of figure 3.17 consists of the set:

$$\{(H, DG), (H, DH), (DG, DH)\}$$

None of the other state-pairs are feasible. Since this machine is rather trivial under the reduction algorithm, we choose a more interesting machine, given in figure 3.21. For this machine, the set of feasible pairs includes all the internal combinations of the sets:

$$\{1, 3\}, \{2, 5, 7\}, \{4\}, \text{and } \{6\}$$

		Input		
	δ	a	b	c
	1	2	5	
	2	3	4	1
	3	5	2	
States	(4)	3	2	1
	5	1	4	1
	6	1		1
	7	3	6	3

Figure 3.21. Another machine to be reduced.

Thus the feasible pairs are

$$\{1, 3\}, \{2, 5\}, \{2, 7\}, \{5, 7\}$$

States 4 and 6 do not appear in any feasible pairs, since they they are distinguishable from all the other states.

We now describe the pairs table construction. Given a pair (p, q) associated with a row, then the table entry for token "a" is the pair (p′, q′), where $p' = \delta(p, a)$, and $q' = \delta(q, a)$. That is, we simply list the pair of states to which (p, q) transfers on each of the input tokens. Note that by the feasible pair selection process, only those states either possessing or not possessing such transitions are in the pairs table; hence we will get either a new state-pair or a null entry. Also, the machine is deterministic, which means that $\delta(p, a)$ and $\delta(q, a)$ contain at most one state each.

The resulting pairs table for the machine in figure 3.21 is shown in figure 3.22(a).

A pair is unordered, e.g., the pair (2, 5) is equivalent to the pair (5, 2). To facilitate the recognition of such equivalences, the members of each pair are written in numeric order in figure 3.22. Thus the states (1, 3) actually transfer to (5, 2) on token "b"; however, the state-pair (5, 2) is written (2, 5). By the same reasoning, if a pair (2, 5) appears in the feasible state-pair list, we do not include (5, 2).

(a) Unmarked

		Input		
		a	b	c
Feasible state pairs	(1,3)	(2,5)	(2,5)	
	(2,5)	(1,3)	(4,4)	(1,1)
	(2,7)	(3,3)	(4,6)	(1,3)
	(5,7)	(1,3)	(4,6)	(1,3)

(b) Marked

		Input		
		a	b	c
Feasible state pairs	(1,3)	(2,5)	(2,5)	
	(2,5)	(1,3)	(4,4)	(1,1)
	✓ (2,7)	(3,3)	(4,6)	(1,3)
	✓ (5,7)	(1,3)	(4,6)	(1,3)

Figure 3.22. Pairs table for machine of figure 3.21.

The next operation is marking those state-pairs that are distinguishable. This is the marking rule: A state-pair in the set of feasible state-pairs is *marked* if there exists a transition to a state-pair (p, q) such that (1) p and q are different, and (2) (p, q) is either marked or not among the feasible state-pairs.

This operation is repeated until no more state-pairs can be marked. Thus in figure 3.22, state-pair (1, 3) transfers to state-pair (2, 5) on both tokens "a" and "b"; (2, 5) is in the feasible state-pair set and is unmarked, hence (1, 3) is not marked.

Consider the pair (2, 5), the second row, which has transitions to pairs (1, 3), (4, 4), and (1, 1). The transitions to (4, 4) and (1, 1) do not call for marking, nor does the transition to (1, 3), since (4, 4) and (1, 1) are singlet pairs, and (1, 3) is in the table, unmarked.

Pair (2, 7) is marked, however, since there is a transition to pair (4, 6), on token "b", and (4, 6) is not in the table. Similarly, pair (5, 7) is marked.

Repeating the operation on pairs (1, 3) and (2, 5), we find that they still are not marked, since (2, 7) and (5, 7) are not among the state-pairs to which they transfer. Hence we conclude that these two pairs remain unmarked, and are therefore the equivalent state-pairs.

The pairs algorithm therefore indicates that the equivalent state-sets of the machine in figure 3.21 are:

$$\{1, 3\}, \{2, 5\}, \{4\}, \{6\}, \{7\}$$

Exercises

1. Reduce the following FSA by the pairs table method:

state	input	
	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

2. Show that the members of each feasible state-pair are 1-equivalent.

3.4. Regular Grammars and FSA

Regular grammars and finite-state automata have a very close correspondence. Given any FSA, a regular grammar may be constructed from it whose language is identical to that of the FSA, and given any regular grammar, an FSA may be constructed from it whose language is identical to that of the

grammar. We give the constructions for these assertions, with examples. The proofs are elementary.

FSA from a Regular Grammar

Recall that a regular grammar $G = (N, \Sigma, P, S)$ has productions in the set P of the form

$A \rightarrow aB$ where $B \in N$, and $a \in \Sigma$
$A \rightarrow b$ where $b \in \Sigma$

Given a grammar G, we construct a NDFSA $M = (Q, \Sigma, \delta, q_0, F)$ as follows:

The states of M are associated with the nonterminals of G, except for one additional state q′ not in N. The halt-state set F is $\{q'\}$. The state q_0 is associated with the start token S in G. Then for every production $A \rightarrow aB$ in G, we add state B to the transition set $\delta(a, A)$. For every production $A \rightarrow b$ in G, we add state q′ to the transition set $\delta(b, A)$.

Note that the alphabets of M and G are identical, and that M may be nondeterministic.

For example, consider the simple grammar G_2, with productions:

$S \rightarrow + N$
$S \rightarrow - N$
$S \rightarrow d N$
$S \rightarrow d$
$N \rightarrow d N$
$N \rightarrow d$

The language of this grammar will be recognized as the signed decimal numbers, where "d" is a decimal digit.

The machine M will contain the states $\{S, N, F\}$, where F is a new halt state.

Then the transitions of M are defined by the following transition function:

$\delta(S, +) = N$
$\delta(S, -) = N$
$\delta(S, d) = N$
$\delta(S, d) = F$
$\delta(N, d) = N$
$\delta(N, d) = F$

The moves of M in recognizing a string x mimic the derivation of x in the grammar. For example, consider the string "-313", or "-ddd" as it appears in the grammar. The moves are

S to N on "−",
N to N on "d",
N to N on "d",
N to F on "d".

The derivation is

$$S \Rightarrow -N \Rightarrow -dN \Rightarrow -ddN \Rightarrow -ddd$$

We leave a proof that L(G) = L(M) to the reader.

FSA to a Regular Grammar

Given a FSA M, a regular grammar G may be constructed from M, such that L(M) = L(G), as follows. Let M be deterministic. Let G = (N, Σ, P, S), where N, the nonterminals, correspond to the machine states Q, the alphabets of G and M are identical, S corresponds to the start state of the machine, and the productions in P are constructed as follows:

- If $\delta(A, a) = B$, then include production $A \rightarrow aB$ in P.
- If $\delta(A, a) = B$, where B is in the halt set F, then include production $A \rightarrow a$ in P.

The resulting grammar is clearly regular. Again, machine M mimics a derivation in G for some string x. We leave a proof that L(M) = L(G) to the reader.

Exercises

1. Construct a regular grammar for the FSA of figure 3.21, for the reduced FSA.

2. Construct an FSA for the regular grammar G, given below. Reduce it and discuss the language L(G) informally.

$A \rightarrow 0B \mid 1D$
$B \rightarrow 0B \mid 1C \mid 1$
$C \rightarrow 1E \mid 0D$
$D \rightarrow 0D \mid 1E \mid 1$
$E \rightarrow 1C \mid 0B$
$F \rightarrow 0C \mid 1G \mid 0$
$G \rightarrow 1E \mid 0F \mid 1$

3. Show that if a string x can be derived in a grammar G, then it is accepted by the FSA M constructed as above. (*Hint*—show by induction on the derivation length).

3.5. Regular Expressions and FSA

A regular expression is a compact way of representing a regular language. In addition to string elements, a regular expression uses three basic operations, *concatenation*, *alternation*, and *closure*. We have introduced each of these operations previously, in a less formal manner.

3.5.1. Definitions

Concatenation is an associative, noncommutative binary operation. The token for concatenation is juxtaposition, e.g., if E_1 and E_2 are two regular expressions, then E_1E_2 is the concatenation of the two. If E_1 and E_2 denote the sets of strings S_1 and S_2, respectively, then E_1E_2 denotes the set

$$S = \{uv \mid u \in S_1 \text{ and } v \in S_2\}$$

Another way to express E_1E_2 is: Choose any string "u" in the set denoted by E_1; choose any string "v" in the set denoted by E_2; then the concatenated string "uv" is in E_1E_2. Also, any string in E_1E_2 can be divided into a "u" prefix and a "v" suffix, where "u" is in the set denoted by E_1 and "v" in the set denoted by E_2.

Alternation is an associative, commutative binary operation, represented by the symbol | or +. If E_1 and E_2 are two regular expressions, denoting the sets of strings S_1 and S_2, then $E_1 \mid E_2$ is a regular expression denoting $S_1 \cup S_2$, the union of S_1 and S_2. We have introduced alternation previously; it is commonly used in the BNF representation of production rules.

Closure is a unary operation. If E is a regular expression denoting some set of strings S, then {E} is a regular expression, called the *closure* of E, and represents the set of all possible strings formed by choosing members of S and concatenating them, together with the empty string ϵ. Thus the closure {E} of a regular expression E is a compact way of writing the infinitely large regular expression

$$\epsilon \mid E \mid EE \mid EEE \mid EEEE \mid \ldots$$

Closure may also be defined by the following set expression:

$$\{E\} = \{\, xY \mid x \in E \text{ and } Y \in \{E\} \,\} \cup \{\epsilon\}$$

This definition may appear circular, since the closure operation which is being defined appears in its own definition. Therefore let us explore this matter briefly. By the definition, the empty string ϵ is in {E}, which means that Y can be ϵ within the definition and, by the first half of the union, all those x's in E are also in {E}. In other words, {E} contains E | ϵ. Thus Y may contain anything in E | ϵ and, by the first half of the union, all those xY's such that x is in E are in {E}; therefore {E} contains EE | E | ϵ, etc.

Closure may also be represented by the token * following the expression to be closed, thus $E^* = \{E\}$. If E consists of more than one token, it must be enclosed in parentheses when this notation is used.

Recall that we earlier stated that Σ^* represents the set of strings in the alphabet Σ. This statement is clearly consistent with the definition of closure.

Parentheses may be used freely in regular expressions as needed to keep the relative ordering of the operations clear. Closure is represented by a parenthesis structure { } and does not require any precedence rule; it operates on the expression contained therein. By convention, concatenation has a higher precedence than alternation, therefore the regular expression

ab|cde

is interpreted

(ab)|(cde)

Closure represented by * has a higher precedence than either concatenation or alternation. Thus

a | bc*

is interpreted

(a) | (b (c)*)

The elements of a regular expression are the tokens of an alphabet Σ, the empty token ϵ, and the null set $\varnothing$. A null-set regular expression denotes an empty set, containing neither ϵ nor any strings. The regular expression ϵ denotes a string set containing only the empty string.

The symbols for alternation, closure, etc. are called *metasymbols* and cannot be in the alphabet of the regular language. Of course, parentheses appear in many common languages, so that any practical implementation must deal with a potential conflict of metasymbols and the language alphabet. A choice of metasymbols might be provided, or a special quote character might be used to delimit alphabet symbols.

A Context-Free Grammar for Regular Expressions

The language of regular expressions is expressed by the following context-free grammar $G_r = (N, \Sigma, P, R)$, where

$$N = \{R, C, L\}; \Sigma = \{+, (,), \{, \}, a\}, \text{ and}$$

the production set P is

$R \rightarrow R + C$ (alternation)
$R \rightarrow C$
$C \rightarrow C\ L$ (concatenation)

$C \rightarrow L$	
$L \rightarrow (R)$	(parenthesizing)
$L \rightarrow \{ R \}$	(closure)
$L \rightarrow a$	(any token in the regular language)
$L \rightarrow \epsilon$	(empty string)

This grammar expresses the precedence of the operations, as well as the structural rules for regular expression formation. In the following discussion, we shall assume that any regular expression may be parsed and a derivation tree created for it.

Examples of Regular Expressions

The regular expression

$$(+ \mid - \mid \epsilon)\ d\{d\}$$

represents the set of (possibly) signed numbers, where d represents the set of digits. This expression may be interpreted in English as follows:

E consists of a choice of "+," "−," or empty (ϵ) followed by a single digit, followed by any number (including zero) of digits.

The regular expression

$$(+ \mid - \mid \epsilon)\ (d\{d\}.\{d\} \mid \{d\}.d\{d\})\ (\epsilon \mid (E(+ \mid - \mid \epsilon)\ d\{d\}))$$

represents a floating point number. To see this, let us divide the expression into three parts as follows:

$$\underset{\text{I}}{(+ \mid - \mid \epsilon)}\underset{\text{II}}{(d\{d\}.\{d\} \mid \{d\}.d\{d\})}\ \underset{\text{III}}{(\epsilon \mid (E(+ \mid - \mid \epsilon)\ d\{d\}))}$$

Then part I represents an optional sign. Part II is the mantissa. It must contain a decimal point and at least one digit ahead or behind the decimal point. Part III represents an optional exponent, signaled by an "E" followed by an optional sign and at least one digit.

Note the use of precedence of concatenation over alternation in part II.

Exercises

1. Describe informally the languages of each of the following regular expressions and give some examples of strings in each one:

$$(A + \{B\})ABB$$
$$00\{01+10+11\} + 11\{01+10+00\}$$
$$(a+b)\{a+b+0\}b$$

2. Construct a regular expression that represents:

   (a) A sequence of 0's and 1's such that the combination 11 appears exactly once.

   (b) Arithmetic expressions containing binary $+$, $-$, $*$, $/$, unary $-$, and one level of parenthesis nesting.

   (c) The set of Cobol identifiers, consisting of a sequence of letters, digits, and underline (_), starting with a letter.

3.5.2. Regular Expression Identities

Regular expressions satisfy a number of identities, which may be used to reduce the complexity of a regular expression or to prove that two regular expressions represent the same language. Unfortunately, there seems to be no systematic procedure for transforming regular expressions into standard forms, as in ordinary algebra or trigonometry.

Let A, B, and C be regular expressions. Then the following identities hold:

(1)	$A + B = B + A$	(commutivity of alternation)
(2)	$\{\varnothing\} = \epsilon$	(closure of an empty set is the null string)
(3)	$A + (B + C) = (A + B) + C$	(associativity of alternation)
(4)	$A(B\ C) = (A\ B)C$	(associativity of concatenation)
(5)	$A(B+C) = A\ B + A\ C$	(distributivity of concatenation over alternation)
(6)	$A\ \epsilon = \epsilon\ A = A$	(identity of concatenation)
(7)	$\varnothing\ E = E\ \varnothing = \varnothing$	(zero of concatenation)
(8)	$\{E\} = E + \{E\}$	
(9)	$\{\{E\}\} = \{E\}$	
(10)	$E + \varnothing = E$	(identity of alternation)

Most of these follow directly from the corresponding properties of the string sets represented by the regular expression. For example, the "+" operator corresponds to a set union, which satisfies commutivity and associativity. Similarly, concatenation can easily be shown to be associative.

Identity 2 follows immediately from the definition of closure– regardless of what (if anything) is in a set E, $\{E\}$ contains ϵ.

Identity 8 follows immediately from the observation that every string in E is also in $\{E\}$, hence the union of these two is exactly $\{E\}$.

Exercises

1. Show that the following are identities through use of the ten identities listed previously.

$$E\{E\} = \{E\}EE$$
$$E(\epsilon + \{E\}E) = \{E\}E$$
$$B(A + AA) = BAA + BA$$
$$\{\{A\}\{B\}\} = \{A + B\}$$

2. Prove each of the ten identities through the use of the equivalent set definitions.

3.5.3. Correspondence to FSA

We now demonstrate that, for every regular expression E denoting some language L(E), there exists a FSA M such that L(M) = L(E). The construction of the machine is particularly useful, since it is often more convenient to represent a language as a regular expression than as a FSA. We therefore need a systematic way to construct a recognizer for the language of a regular expression.

The machine we construct will be nondeterministic, in general, but may be reduced to a minimal deterministic machine by the methods of the previous sections.

The basic idea of the transformation is simple. We conceive a FSA that contains a start state S and one halt state F. Somehow, it recognizes a regular expression E, as diagrammed in figure 3.23(a). The square box containing "E" represents a set of states and transitions between states S and F.

Now suppose that E is the empty set $\varnothing$. Since $\varnothing$ is an empty language, which contains neither any string nor ϵ, the only possible machine for $\varnothing$ is the isolated S and F states, figure 3.23(b). This machine refuses to accept any string, including the empty string, since it can never reach the halt state.

Next suppose that E is the empty string ϵ. A recognizing machine for this language is shown in figure 3.23(c), which permits one transition—an empty move from S to F. This machine clearly exactly accepts the empty string. A string of length > 0 is rejected, since there are no other moves from either S or F.

If E is an alphabet token "a", the machine of figure 3.23(d) exactly recognizes a string consisting of that token.

Now we consider some more sophisticated machines. If E is a parenthesized expression, E = (E′), a recognizer machine for E is clearly a recognizer for E′.

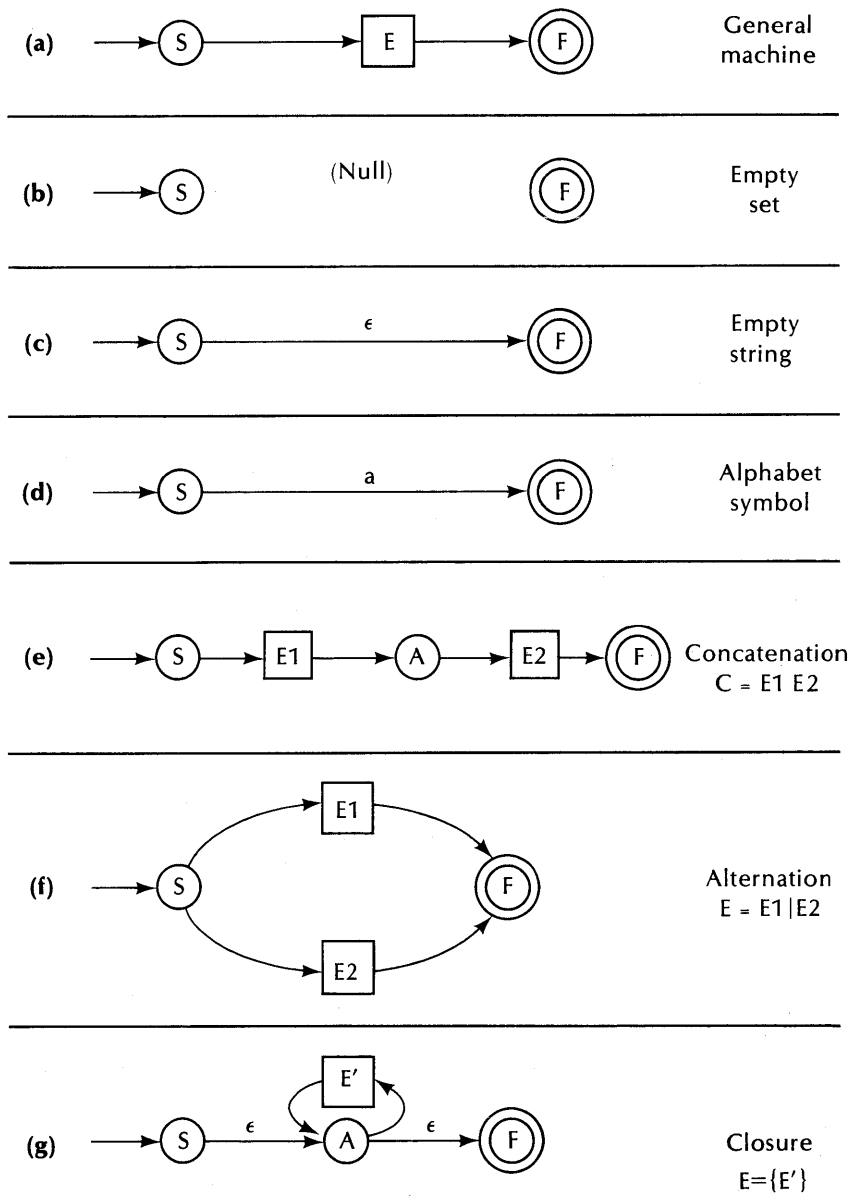


Figure 3.23. Generation of a non-deterministic finite-state automaton from the components of a regular expression.

Consider the concatenation E of two expressions, E1 and E2. A recognizer for E clearly must recognize E1, then E2, in that order. Such a machine is shown in figure 3.23(e).

An alternation expression $E = E_1 \mid E_2$ is recognized exactly by the machine of figure 3.23(f). Recall that the set of strings represented by $E_1 \mid E_2$ is the union of the sets represented by E_1 and E_2. Now let machine M_1 recognize E_1 and M_2 recognize E_2. Then consider a string x in E. The string x must be in E_1 or in E_2 (it could belong to both, of course). If in E_1, then the upper path of figure 3.23(f) yields a recognition; if in E_2, then the lower path of figure 3.23(f) yields a recognition. No strings other than those in the union of $L(M_1)$ and $L(M_2)$ are in the composite machine; any string recognized by the composite would have to be recognized in the upper or lower path, therefore by either M_1 or M_2.

Finally, a closure expression $E = \{E'\}$ is recognized exactly by the machine of figure 3.23(g). The empty moves are needed when these machine segments are combined to form a complete recognizer for some regular expression. Machine (g) clearly recognizes the empty string (two empty moves, S to A, A to F), and one or more concatenations of the strings in E'. Thus a string consisting of n members ($n \geq 0$) of the regular expression E'may be recognized by the empty move from S to A, n moves from A to itself through the machine for E', followed by the move from A to F.

Figure 3.23 essentially outlines the rules by which a complete machine for an arbitrary regular expression may be built from its parts. The construction operations are effectively guided by the derivation tree for the regular expression, and may be done bottom-up or top-down. The top-down process will be be illustrated for an example. Then a top-down procedure will be given.

Consider the regular expression

$$E = (+ \mid - \mid \epsilon)d\{d\}$$

which displays all the operations and tokens of regular expressions except the $\varnothing$ set, which should never appear within a regular expression anyway.

A simplified tree for this expression is given in figure 3.24. It is essentially the derivation tree with the single productions and parenthesis nodes removed.

At the root level, E consists of a concatenation of two pieces, which we shall call E_1 and E_2:

$$E = E_1\ E_2$$

where $E_1 = (+ \mid - \mid \epsilon)$ and $E_2 = d\{d\}$. Thus the first machine looks like figure 3.25(a). We have introduced a new intermediate state, A.

The transition from S to A is an alternation of + with $(- \mid \epsilon)$, as shown in figure 3.24. The single transition is therefore split into two, one for "+", the other for "$(- \mid \epsilon)$". The latter in turn splits into one for "−" and one for "ϵ".

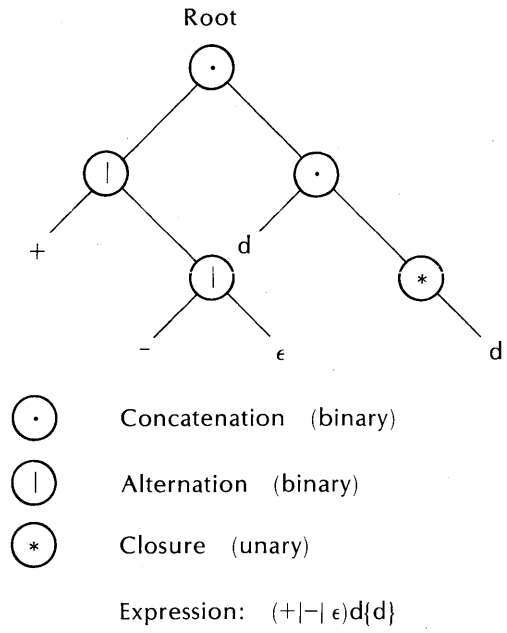


Figure 3.24. The regular expression "(+|−| ϵ)d{d}" as a tree.

The resulting machine is shown in figure 3.25(b).

Turning to the machine between states A and F, we see that it is another concatenation, of "d" with "{d}". The machine of figure 3.25(c) is the result, containing another new state, B.

The last machine is between states B and F and is a closure machine. The final NDFSA for the regular expression is therefore shown in figure 3.25(d). As it turns out, neither of the ϵ-moves between states B and F are necessary in this machine; however, there is no harm in keeping them, since they will be eliminated and the machine reduced to minimal form by the methods previously described.

We now give a recursive procedure FSM(EXPR, P, Q), which, when given a regular expression EXPR, an initial state P = S, and a final state Q = F, yields a set of states and a transition function for a machine (nondeterministic in general) whose language is that of the regular expression. The final machine has one halt state, F.

We use a Pascal-like notation for the procedure. The operator "=" is an equality comparison; operator ":=" means the left side's value is replaced by the value of the right-side expression. Lowercase words represent keywords, and uppercase words represent variables and arguments. The "∪" is the set union operation. A comment starts with { and ends with }. The variable δ represents the tabular representation of the transition function. Thus

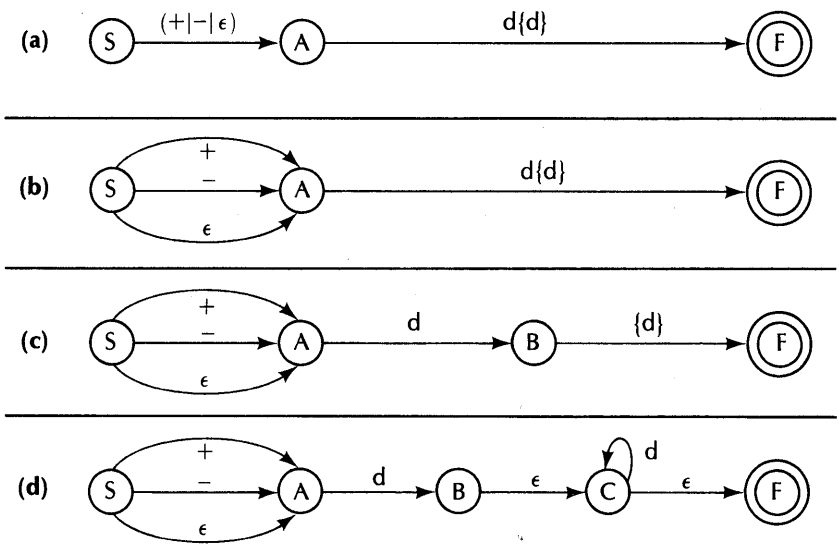


Figure 3.25. Development of a finite-state automaton from the regular expression "(+|−|ϵ)d{d}".

$$\delta(R, A) := \delta(R, A) \cup \{Q\}$$

means that state Q is added to the set δ(R,A).

```
procedure FSM(EXPR, P, Q);
begin
    if EXPR = ∅ then return;
    if EXPR = ϵ then
    begin
        δ(P, ϵ) := δ(P, ϵ) ∪ {Q};
        return;
    end;

    if EXPR = a ϵ Σ, then
    begin
        δ(P, a) := δ(P, a) ∪ {Q};
        return;
    end;

    if EXPR = (X) then
    begin
        call FSM(X, P, Q);
```

```
        return;
    end;
    if EXPR = X Y then    {concatenation}
    begin
        create a new state A;
        call FSM(X, P, A);
        call FSM(Y, A, Q);
        return;
    end;
    if EXPR = X | Y then  {alternation}
    begin
        call FSM(X, P, Q);
        call FSM(Y, P, Q);
        return;
    if EXPR = {X} then    {closure}
    begin
        create a new state A;
        call FSM(ϵ, P, A);
        call FSM(X, A, A);
        call FSM(ϵ, A, Q);
        return;
    end;
end;
```

Example. Let EXPR = (+ | − | ϵ)d{d}. Then we call FSM(EXPR, S, F). S will be the start state and F the (only) halt state of the final machine for EXPR. The following is a trace of the calls of FSM, the actions taken, and the returns. The periods "..." indicate the depth of nesting in the recursive calls.

```
call FSM( '(+ | − | ϵ)d{d}', S, F);
. create a new state: R1;
. call FSM( '(+ | − | ϵ)', S, R1);
. . call FSM( '+ | − | ϵ', S, R1);
. . . call FSM( '+', S, R1);
. . . . δ(S, +) := δ(S, +) ∪ {R1};
. . . . return;
. . . call FSM( '− | ϵ', S, R1);
. . . . call FSM( '−', S, R1);
. . . . . δ(S, −) := δ(S, −) ∪ {R1};
. . . . . return;
```

. . . . call FSM('ϵ', S, R_1);
. . . . . $\delta(S, \epsilon) := \delta(S, \epsilon) \cup \{R_1\}$;
. . . . . return;
. . . . return;
. . . return;
. . return;
. call FSM('d{d}', R_1, F);
. . create a new state: R_2;
. . call FSM('d', R_1, R_2);
. . . $\delta(R_1, d) := \delta(R_1, d) \cup \{R_2\}$;
. . . return;
. . call FSM('{d}', R_2, F);
. . . create a new state: R_3;
. . . call FSM('ϵ', R_2, R_3);
. . . . $\delta(R_2, \epsilon) := \delta(R_2, \epsilon) \cup \{R_3\}$;
. . . . return;
. . . call FSM('d', R_2, R_3);
. . . . $\delta(R_2, d) := \delta(R_2, d) \cup \{R_2\}$;
. . . . return;
. . . call FSM('ϵ', R_2, F);
. . . . $\delta(R_2, \epsilon) := \delta(R_2, \epsilon) \cup \{F\}$;
. . . . return;
. . . return;
. . return;
. return;

The result of this FSM call is the transition function shown graphically in figure 3.25(d).

Exercises

1. Show that the regular expression

$$(+ \mid - \mid \epsilon)\,(d\{d\} \mid d\{d\}.\{d\} \mid \{d\}.d\{d\})$$

   yields the FSA of figure 3.3.

2. Add an exponent part to the expression of the previous expression, develop its FSA and reduce it.

3. Define data structures for procedure FSM in Pascal and develop a complete program around it. Assume that a reduced tree structure for the regular expression is available, similar to that in figure 3.24. (The next chapters deal with the problem of parsing a regular expression and generating a reduced tree from the parse).

3.5.4. Regular Expression of a Regular Grammar

We first observe that a given regular language has many representations in regular expressions, and that it is not easy to reduce a given regular expression to some minimal form. The method given below always yields a valid regular expression, but it may be larger than another one that also represents the same language. We know of no systematic reduction process for regular expressions similar to those useful in ordinary algebra.

We first introduce the concept of a *regular expression equation*, which contains regular expressions and variables X[1], X[2], etc., that stand for some unknown regular expression. These resemble linear equations and are written in standard form as follows:

$$X[1] = a[1,0] + a[1,1]\,X[1] + a[1,2]\,X[2] + \ldots + a[1,n]\,X[n]$$

$$X[2] = a[2,0] + a[2,1]\,X[1] + a[2,2]\,X[2] + \ldots + a[2,n]\,X[n]$$

$$\vdots$$

$$X[n] = a[n,0] + a[n,1]\,X[1] + a[n,2]\,X[2] + \ldots + a[n,n]\,X[n]$$

Each of the coefficients a[i,j] is a regular expression in general, but contains no variables.

Note that a set of productions in a regular grammar may be represented as a set of regular expression equations. For example, the regular grammar

$$\begin{aligned}
S &\rightarrow 0A\\
S &\rightarrow 1B\\
S &\rightarrow 0\\
S &\rightarrow 1\\
A &\rightarrow 0S\\
A &\rightarrow 1B\\
A &\rightarrow 1\\
B &\rightarrow 0A\\
B &\rightarrow 1S\\
B &\rightarrow \epsilon
\end{aligned}$$

may be written as a set of three regular expression equations in the three unknowns, S, A, and B, as follows:

$$S = (0+1) + 0A + 1B \tag{3.1}$$
$$A = 1 + 0S + 1B \tag{3.2}$$
$$B = \epsilon + 1S + 0A \tag{3.3}$$

Thus, the first four productions are equivalent to

$$S \rightarrow 0A \mid 1B \mid 0 \mid 1$$

which may be written, using "+" for the alternation "|" and "=" for "→", as:

$$S = 0+1+0A+1B$$

We then observe that if it is somehow possible to solve a system of equations for the variable S, the solution being a regular expression in the alphabet, we will have a regular expression representing the language of the underlying grammar.

We first need a solution for the equation

$$S = aS + b \tag{3.4}$$

where "a" and "b" are regular expressions in the alphabet and possibly in the other variables. Note that any regular expression equation in any variable S may be written in this form.

A solution for this equation is

$$S = \{a\}b$$

or $S = a^* b$, using the alternate notation.

To prove this, consider the substitution of $\{a\}b$ for S in equation (3.4):

$$\{a\}b = a\{a\}b + b \tag{3.5}$$

Factor the right side, yielding

$$\{a\}b = (a\{a\} + \epsilon)b$$

Now $\{a\} = a\{a\} + \epsilon$, since ϵ is in both sides, "a" is in both sides, and any string in $\{a\}$, other than "a" or ϵ, is in $\{a\}$ and in $a\{a\}$. Hence Eq. (3.5) is an identity, and $\{a\}b$ is a solution of Eq. (3.4).

We will not show that $\{a\}b$ is in some sense a complete solution of Eq. (3.4). Now there are solutions to (3.4) that are not in $\{a\}b$, if "a" contains the empty string. Indeed, $\{a\}(b+c)$ is a solution to (3.4), where c represents any set of strings whatsoever, if "a" contains the empty string. However, it turns out that $\{a\}b$, called the *minimal fixed point* of Eq. (3.4), is sufficient to generate an equivalent regular expression.

Now we can solve a general system of equations for the start token S. We illustrate the method using Eqs. (3.1) to (3.3) given above. The general method should be clear from this example; a more rigorous treatment is given in Aho[1972].

We start with some equation other than the S equation, for example, the B equation, Eq. (3.3). If this had the form

$$B = aB + b$$

we would first transform it into the equation

$$B = \{a\}b$$

which eliminates B from the right-hand side. Since the right side of (3.3) does not contain B, this step is unnecessary.

The regular expression obtained for B, which is just Eq. (3.3), may now be substituted into the other equations. The resulting equations are then free of variable B. The result of this substitution in the set (3.1) and (3.2) is

$$A = 1 + 0S + 1(\epsilon + 1S + 0A)$$

$$= 1 + (0 + 11)S + 10A \tag{3.5}$$

$$S = (0 + 1) + 0A + 1(\epsilon + 1S + 0A)$$

$$= (0 + 1) + (0 + 10)A + 11S \tag{3.6}$$

We have made use of some of the identities in the second step in each case. For example, in the A equation,

$$1 + 0S + 1(\epsilon + 1S + 0A) = 1 + 0S + 1\epsilon + 11S + 10A$$

$$= 1 + 1 + 0S + 11S + 10A = 1 + (0 + 11)S + 10A$$

in Eq. (3.5).

We next rewrite Eq. (3.5) in the form $A = aA + b$:

$$A = 10A + (1 + (0 + 11)S) \tag{3.7}$$

which has the minimal fixed point solution:

$$A = \{10\}(1 + (0 + 11)S) \tag{3.8}$$

Substituting this solution into the remaining S equation yields:

$$S = (0+1+(0+10)\{10\}1) + ((0+10)\{10\}(0+11)+11)S \tag{3.9}$$

after some rearrangement and factoring. It has the fixed-point solution

$$S = \{(0+10)\{10\}(0+11)+11\}(0+1+(0+10)\{10\}1) \tag{3.10}$$

which should be a regular expression equivalent to the original regular grammar given above. As a check, it would be wise to construct an automaton from Eq. (3.10), reduce it, and verify that its regular grammar agrees with the original grammar. It is clearly not obvious that Eq. (3.10) reflects our grammar, nor is it clear whether a shorter expression can be found for S. Different expressions result, depending on which variables are eliminated first, and it may pay to do the reduction in different ways to see if a shorter expression can be obtained.

Exercises

1. Transform the FSA of figure 3.25(d) into a regular grammar, then into a regular expression. Can you show that the result is equivalent to (+ | − | ϵ)d{d}? (*Note*: Empty moves must be removed first.)

2. Solve the following set of regular expression equations:

$$\begin{aligned} A &= (\{0\}+1)A + B \\ B &= 11 + 0A + 11C \\ C &= \epsilon + A \end{aligned}$$

3. Show that {a}(b+c) is a solution of S = aS+b if "a" contains ϵ, and "c" is any string whatever.

3.6. FSA Representations

A deterministic FSA may be embedded in a computer program in either of two ways—as a set of tables which are interpreted by a general purpose program, or as a specially constructed program that represents the machine. The table approach is usually superior to the program approach for large automata in memory and in reliability. The table interpreter need be written only once for any machine whatever, and usually a machine table requires less storage space than the equivalent program instructions. However, interpretation of a table demands more running time. Therefore, if minimal run-time without regard to storage space is wanted, a program approach is better. If minimal storage space is wanted, then the table approach is better.

Interpreted Tables

A FSA table is based on its transition table. Usually, semantic actions on certain transitions are also needed, hence each transition should carry an additional table entry that specifies a semantic action.

FSA tables are generally sparse. By this we mean that most of the table entries are empty. For a large machine, over 90 percent of the table may consist of empty entries. We therefore have an opportunity to construct a set of tables that specify only the useful entries, omitting the empty entries. We now describe how this might be done.

3.6.1. Sparse Array Tables

The fundamental idea of a sparse array table system is rather simple, although the implementation looks complicated. We collect all the non-

NULL table entries together in order into one linear array, called VALUEA. Let its size be T; this is the total number of non-NULL entries in the original table. Then we create another array of size T. It carries one of the index values associated with the values in VALUEA, say the J indices. Call this array INDEXJ. Now suppose the value V (non-NULL) is in the original array at position (I,J), and shows up in the VALUEA array at index K. Then INDEXJ(K) = J.

We now need a guide to the I index values in VALUEA and INDEXJ, and it is provided by two other smaller arrays, INDEXI and NUMBEROFJ. Given an index I, INDEXI(I) is the VALUEA and INDEXJ index of the list found in row I of the original matrix. NUMBEROFJ(I) is the number of values in that list.

For example, consider the following matrix A(I,J):

I \ J	1	2	3	4	5
1				15	9
2	3				7
3		5			
4				10	

We first list its values in the array VALUEA, working left-to-right then down:

K	VALUEA
1	15
2	9
3	3
4	7
5	5
6	10

Next, we add the associated INDEXJ values:

K	VALUEA	INDEXJ
1	15	3
2	9	4
3	3	1
4	7	5
5	5	2
6	10	4

The last tables are INDEXI and NUMBEROFJ, as follows:

I	INDEXI	NUMBEROFJ
1	1	2
2	3	2
3	5	1
4	6	1

Then, to find A(2, 5), we enter INDEXI(2) and NUMBEROFJ(2), yielding 3 and 2, respectively. Then enter INDEXJ(3) and search the list for at most 2 items; these are 1 and 5. Since 5 matches J in A(I, J), the associated VALUEA is 7, which is A(2, 5).

The algorithm is expressed by the following Pascal procedure, that returns the value A(I, J), given the array declarations described above:

```
function A(I, J: integer): integer;
begin
    var K: integer;
    if NUMBEROFJ(I)=0 then A:=0  {indicates empty}
    else
    begin
        for K:=INDEXI(I) until
                INDEX(I)+NUMBEROFJ(I) do
        begin
            if J=INDEXJ(K) then
            begin
                A:=VALUEA(K);
                return
            end
        end;
        A:=0
    end
end
```

Suppose an array A has the dimensions (M, N), and it contains T useful (nonempty) elements. Then a two-dimensional array would require $M \times N$ entries, while the reduced sparse matrix system defined above would contain $2(M+T)$ entries. There is a considerable saving in storage space with the sparse scheme for large M and N, and T much smaller than $M \times N$. For example, if $M=20$, $N=100$, and $T=20$, then $M \times N = 2000$, while $2(M+T) = 80$, a reduction in table size of 25 to 1. Furthermore, the NUMBEROFJ array usually consists of fairly small numbers, which may possibly be compacted in memory as bytes, which the direct storage of an array A requires $M \times N$ units of whatever storage is required for the A values.

Clearly, either of the dimensions in A(I, J) may be chosen for the INDEXI table. The sizes of the INDEXJ and VALUEA tables are not affected by the choice, but the INDEXI and NUMBEROFJ tables are. The index (I or J) with the least cardinality should therefore be selected for the INDEXI and NUMBEROFJ table index.

3.6.2. Table Reductions

The NUMBEROFJ table is often unnecessary. If INDEXI is arranged in monotonic increasing order, we may infer the NUMBEROFJ value from two consecutive INDEXI values. That is,

$$\text{NUMBEROFJ(I)} = \text{INDEXI(I+1)} - \text{INDEXI(I)}$$

which is valid for all but the largest I. We therefore need one more entry in INDEXI, for index 5, and the NUMBEROFJ table may be dispensed with. The resulting INDEXI table from the previous example then looks like this:

I	INDEXI
1	1
2	3
3	5
4	6
5	7

The INDEXJ list may also be arranged in monotonic increasing order. The storage space required for it may be reduced appreciably in one of two ways. Suppose the smallest unit of storage is an element whose maximum value is V. Then the state table may be listed as actual states, until the state number exceeds V. At that point, all of the subsequent numbers are reduced by V, and started over. A special marker is needed to indicate the "breaks" in the table, and the table search algorithm must be organized to reflect this change.

Another way to reduce the INDEXJ list is to record only the positive increments from one state to the next. Except for the first state, which could be a fairly large number, most of the subsequent states are likely to be small numbers. This plan breaks down if the table contains a few large increments.

3.6.3. Sparse Array Representation of a FSA

A FSA is clearly expressed by its transition function, which is just a two-dimensional array, $\delta(p, a)$. Following the guidelines given above, we should choose the least dimension. Usually the token set is smaller than the state set, particularly if letters can be lumped together into one token and numbers can be lumped together into one token, for transition purposes.

Then the total number of tokens may be a few dozen at most. The number of states can be much larger, and depends on the complexity of the machine's language.

We also need some way to indicate semantic actions and whether the machine may halt in a given state. We are therefore led to the following set of tables for a general FSA:

READX, size = number of tokens. For a token I, READX(I) is an index into the STATE table.

STATE, size = number of transitions. Let K = READX(I), for an input token I, and N = (READX(I + 1) − READX(I)). Then the list STATE(K), STATE(K + 1), . . . , STATE(K + N − 1) contains all the legal present states corresponding to the input token I. If a match with the present state S cannot be found, then the machine blocks (error). If a match can be found, let the STATE index be M, i.e., STATE(M) = present state.

CALL, size = number of transitions. CALL(M), where M is found as explained in the preceding paragraph, contains information on which semantic action to perform. In general, CALL(M) will be a number specifying one of several possible semantic actions, a branch address, or a subroutine location.

HALT, size = number of transitions. This array contains single bits, T or F. If HALT(M) = T, then a halt is legal on this state and input token; otherwise, a halt is illegal (error if this is the last token).

GOTO, size = number of transitions. This array contains the next state; i.e., if the current state P and token "a" result in the STATE index M, then GOTO(M) = δ(P, a). The next state is used if another input token exists; otherwise, the machine halts and reports "error" or "accept", depending on the HALT bit.

Example FSA Expressed as a Sparse Table

Figure 3.26 shows the FSA of figure 3.17 expressed as a sparse matrix system. (State DH is equivalent to state DG, and is therefore not included.) The states are numbered:

S:	1
A:	2
G:	3
H:	4
BCE:	5
DG:	6

Index	Symbol	State index
1	(+)	1
2	(−)	2
3	(.)	3
4	(d)	6
5		12

READX

Index		State		Call	Halt	GOTO
(+)	1	1	(S)	1	F	2
(−)	2	1	(S)	2	F	2
(.)	3	1	(S)	3	F	3
(.)	4	2	(A)	4	F	3
(.)	5	5	(B,C,E)	5	T	6
(d)	6	1	(S)	6	F	5
(d)	7	2	(A)	7	F	5
(d)	8	3	(C)	8	F	4
(d)	9	4	(H)	9	T	4
(d)	10	5	(B,C,E)	10	T	5
(d)	11	6	(D,G)	11	T	6

Figure 3.26. Finite-state automaton expressed as a sparse matrix table, for machine of figure 3.17.

The input tokens are also numbered:

+:	1
−:	2
.:	3
digit:	4

For example, consider input token "digit" (4), and present state H (4). The READX table says that the list of legal current states begins at index 6 in STATE table, and the list contains 6 elements. We therefore search the STATE table from index 6 through 11, and find state 4 at index 9. Corresponding to this index is a semantics operation call (9), a "T" HALT indicator (legal to halt if this is the last token), and GOTO = 4 (next state is number 4, H).

Exercises

1. Trace the acceptance of the string

   +dd.d

   through the sparse table system of figure 3.26.

2. Write a Pascal procedure that interprets a set of reduced FSA sparse tables and calls an action procedure on every transition with a semantics action indicated in the table. It accepts tokens from a scanner.

3. Write a program that generates reduced tables from a complete state table.

3.6.4. Program Representation of a FSA

A direct program representation of a FSA will be the most time-efficient means of representation, if memory size is no consideration. There are many forms of representation, and the best will likely depend on the kind of machine instructions available. If the machine contains an instruction that can search for a given number in a list of numbers, returning its index, then essentially it can solve the problem of searching the STATE list in the sparse tables described above, and may also be used effectively in a direct program representation.

For example, the following program represents the FSA of figure 3.1. The states S, A, B, G, and H are represented by numbers 0, 1, 2, 3, and 4. It contains several utility procedures, as follows:

OPEN'INPUT, which opens the input list file, preparing it for reading by procedure NEXT'CHAR.

NEXT'CHAR, which fetches the next character, placing it in CHARACTER, returning TRUE if a character exists or FALSE if the input list is exhausted.

ERROR, which reports a machine block or failure to terminate in a halt state. What happens after an error is detected is not defined in this program. We may assume for now that the error is simply reported, and the program halts.

FSA Program Example

```
STATE:=0;
OPEN'INPUT;
```

```
while NEXT'CHAR do
begin
   case STATE of
   0: if "0" ≤ CHARACTER ≤ "9" then STATE:=2
      else if CHARACTER = "+"
              or CHARACTER = "−" then
           STATE:=1
      else if CHARACTER = "." then STATE:=3
      else ERROR;
   1: if "0" ≤ CHARACTER ≤ "9" then STATE:=2
      else if CHARACTER = "." then STATE:=3
   2: if "0" ≤ CHARACTER ≤ "9" then STATE:=2
      else if CHARACTER = "." then STATE:=4
      else ERROR;
   3: if "0" ≤ CHARACTER ≤ "9" then STATE:=4
      else ERROR;
   4: if "0" ≤ CHARACTER ≤ "9" then STATE:=4
      else ERROR;
   end;
end;    {end of while-do loop}
    {if here, the input list is empty}
if not(STATE=2 or STATE=4) then ERROR;
```

Summary

The choice between a direct program representation or an interpreted table approach to a FSA depends on the size of the FSA, the desired efficiency, and the size of the available memory. For a small FSA, containing perhaps fewer than a dozen or so states, a direct program representation would probably be more efficient and require less computer memory, since the interpreting program is fairly complicated, and is no different in memory size for a large FSA than for a small one.

For a large FSA, a direct program should be attempted only if an automatic program generator is available. The possibility of an error increases with the number of transitions, and the program itself becomes less and less readable. Maintenance is also a problem, since a minor change in the FSA may require a major rewriting of the program and all the state numbers.

With a table approach, the memory size is minimal, once the interpreting program space is considered. A change in the FSA requires only a table change and a few changes in the affected semantic procedures, nothing else. An automatic FSA generator should clearly generate reduced sparse tables that can be immediately used in a general-purpose FSA interpreting program.

Exercise

Propose other program-oriented FSA representations, including one in assembly language for your favorite machine architecture.

3.7. Applications of FSA

FSA have many applications in compilers. The parser in an LR(1) recognizer contains a FSA (section 2.3.4). Certain symbol table operations are nicely modeled as a FSA (section 7.4). We shall explore some of the concepts of lexical analysis in the remainder of this chapter.

3.7.1. Recognition of Literals

Most common programming languages contain a class of literal constants that can be recognized by a finite-state machine. They may appear in the compiler source language and in data to be formatted.

Let us consider one example, based on the automaton studied in the first part of this chapter. The design of a recognizer may start with a deterministic or nondeterministic automaton, a regular expression or a regular grammar. The choice depends on the nature of the literal specification in the language. If the specification is in BNF, it may be best to attempt to construct a regular expression from the BNF by the expansion methods of section 3.5. In any case, the end result of a definition and a reduction must be a deterministic FSA, such as the one in figure 3.1 for simple decimal numbers.

Let us add conversion operations to this machine. In general, we assign one or more registers to the machine, and then assign operations on the register contents to each of the transitions. The result is a machine as shown in figure 3.27. There are three registers, SIGN, P, and N. SIGN holds either -1 or $+1$. The registers P and N hold floating-point numbers. Certain of the transitions carry operations on these registers. We include transitions into the start state and out of the halt states for completeness.

Initially, SIGN is $+1$, P is 10.0, and N is 0.0. SIGN is changed to -1 only upon the transition S to A on "$-$"; otherwise it remains $+1$. Each transition

on a digit "d" involves the function "value(d)"—the numeric equivalent of the character "d". Thus in the S to B transition, value(d) is assigned to N. Upon an exit from B or H, the final value of the number is SIGN*N.

On the transition B to B, the old value of N is multiplied by 10 and added to value(d). The digit d precedes any decimal point, hence each digit causes the place value of those preceding them to be increased by a factor of 10.

On scanning a decimal point, the register P is used; P carries the power of ten corresponding to the place value of the digits following a decimal point. Thus in the transition G to H, value(d)/P is added to N, and P is increased by a factor of 10.

An exit from halt state B clearly means that no decimal point was scanned; the number is an integer. The exit from halt state H means the number carries a decimal point.

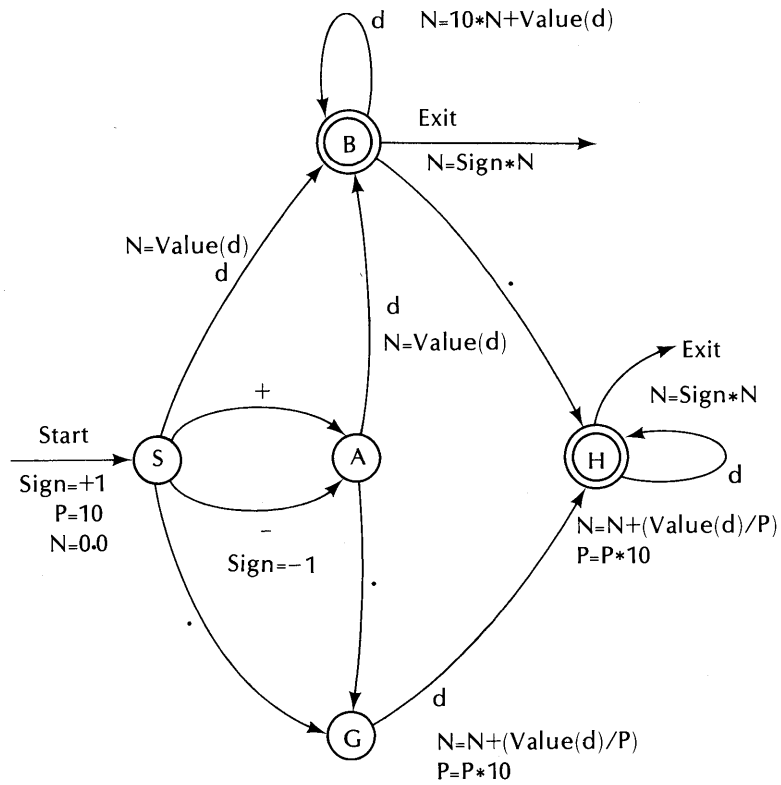


Figure 3.27. The machine of figure 3.1 with semantic operations added. This machine recognizes a decimal number and converts it to an internal form.

The exits from B and H are clearly taken on any token other than "." or "d" for B or "d" for H. A *lookahead* of one token is evidently needed in order to exit properly. However, there may be other kinds of halt indicator. If the recognizer is used in a Fortran formatter to recognize numbers in an input record with fixed column boundaries, it may be necessary to halt on the end of the field, not the next character. Such details can clearly be worked out as a programming exercise.

Exercises

1. Trace figure 3.27 with the string "−33.62".
2. Extend figure 3.27 to exponent parts for floating point numbers.
3. Discuss optional blanks in a language: (a) If a number can be preceded by any number of blanks, what additional states are needed in the FSA of figure 3.27? (b) If a number can contain blanks separating any of its parts except the digits in the numbers, what additional states and/or transitions are needed?
4. Construct a FSA that accepts either numbers, Fortran identifiers, quoted strings, or comments. The quoted strings are delimited by a pair of quote marks, and may contain a quote mark by writing it twice. A comment begins with the keyword COMMENT and ends on a semicolon. Invent something to deal with the problem that COMMENT is in the class of Fortran identifiers.
5. Make an assertion that holds for each of the states in the FSA of figure 3.27. Use these states to prove that the FSA correctly generates the numeric equivalent of accepted input strings. For example, a reasonable assertion for any of the states is, "N is the absolute value of the decimal number scanned so far."

3.7.2. Lexical Analysis

A *lexical analyzer* assembles sequential groups of one or more characters in the source into tokens. The process may be relatively simple or quite difficult, depending on the language.

A lexical analyzer comprises a number of functions:

1. Source file handler. The lexical analyzer is responsible for opening and reading the source file. Some compilers permit several input files to be merged, or for a source file to call out another file, etc. Record

boundaries must either be made into tokens or made invisible to the parser.

2. Comment scanner. Most programming languages contain provisions for comments, which may appear anywhere and are therefore difficult to build into a grammar. The lexical analyzer must then recognize them and make them invisible to the parser.

3. Macro processor. A macro processor is essentially a string expander that looks for macro definitions and macro calls and expands the latter. Usually, the macro processor has little or no relation to the syntax of the source program that it generates, and it may therefore process the source independently of the rest of the compiler.

4. Token assembler and screener. The lexical analyzer is responsible for collecting sequences of characters in the source into identifiable tokens for the parser. The tokens it assembles correspond to the terminals of the language's grammar. The recognition of tokens may not be a trivial matter, as some programming languages contain keywords that can also be used as variable names.

5. Literal converter. Floating point literals, fixed point literals, literal strings, and other literal forms may all be recognized and converted to internal form by the lexical analyzer.

Source Records and Characters

A programming language may be either free-form or line-oriented. Algol 60, Algol 68, Pascal, and PL/I are examples of free-form languages; Basic and Fortran are examples of line-oriented languages.

In a free-form language, the characters comprising the source are conceptually assumed to be one long sequence uninterrupted by source record boundaries. Thus if the physical records of the source happen to be 80 characters each, some procedure in the compiler must arrange that the 80th character of each record is effectively contiguous with the first character of the next record; that is, the record boundaries must be invisible to the lexical analyzer and the parser. The lexical analyzer is best organized in two procedural levels: GETCHAR and GETTOKEN, as follows:

GETCHAR is at the lowest level, is called by GETTOKEN, and supplies one character on each call. It is responsible for accessing the source file and for processing any special control records that are not in the compiler's language. For example, in many large systems, records with certain special characters in the left-most columns must be recognized as control records, with a special syntax.

GETTOKEN calls GETCHAR for each character needed in assembling a token. It knows nothing of the source record arrangement, but must be tailored to the language. GETTOKEN is called by the parser system and supplies one token for each call.

In a line-oriented language, the parser and scanner system may be reset to some known starting state at the beginning of each line. Line boundaries may or may not serve as delimiters, however, the beginning of the next line certainly serves as an end-of-line delimiter. The Fortran conventions are interesting and illustrate a class of lexical analyzer problems. The lines are organized as 80-byte records, corresponding to a standard IBM card. Column 1 is used to indicate that the record is a comment. A statement label number may appear in columns 2 to 5. Column 6 is not used on the first of several continuation records. The source text resides in columns 7 to 72. If column 6 of the next record is marked, it serves as a continuation record, in which case, column 7 of the second record must effectively follow column 72 of the first (the boundary is invisible). Up to 19 continuation records are permitted. Statement labels are not permitted on continuation records.

A Fortran statement therefore has the form

<label> <delimiter> <statement> <end-of-statement>

where <delimiter> and <end-of-statement> are special tokens that separate the statement label, the statement, and the next record.

A *token* is a member of the terminal alphabet of the grammar and may be a single character or one of the following common character sequences:

Two or more special characters, e.g., {:=, <=, **}.

Identifier. Generally an identifier begins with a letter and continues with letters, digits and (sometimes) certain special characters, e.g., PAY_ROLL (Cobol), or S156 (most common languages). Identifiers must usually be classified as *user names* and *keywords*. A user name is invented by the programmer to serve as a data, procedure, or statement label. A keyword is a terminal token in the language's grammar that happens also to fit an identifier pattern. In Pascal, WHILE, DO, IF, THEN, etc. are keywords. The task of separating keywords and user names is often difficult.

Number. The number formats of the language often include fixed- and floating-point forms, e.g., -17.56E-22 (Fortran). In most compilers, the lexical analyzer recognizes and converts such numbers. However, a complicated number format may also be easier to recognize through the parser system, so that the lexical analyzer need only return the smaller tokens "17", ".", "E", etc.

Quoted string. Many programming languages permit an arbitrary sequence of characters to be quoted and treated as a single token, a *string*.

Counted string. One of the Fortran FORMAT statement forms is a string preceded by a count, e.g., 5H# #)ab. The string following the "H" can only be determined by first scanning the count "5" and then counting off five characters; this is a lexical analyzer task.

Distinguishing Tokens

Algol 60 contains each of the following tokens:

: used between a statement label and the statement.
= used as an "equal" relation.
:= used as the replacement operator.

The lexical analyzer must distinguish these, and can do so by a left-to-right scan of the source characters, along with a single character lookahead. If ":" has been scanned, and the next character is not "=", then the returned token must be ":". In general, the analyzer should assemble the longest sequence of characters that is consistent with a single token in the language. Whether this rule will be effective must be investigated through a study of the grammar. The grammar should be such that token "=" can never follow ":" except in the composite token ":=". Other composite tokens must satisfy a similar property. A discussion of the *follow* problem is given in the next chapter.

A *delimiter* is any character that serves to terminate a token; it is not a part of the terminated token, but may itself be a token or part of a token. In some languages, a blank is a valid delimiter. In others, Fortran in particular, a blank is ignored except in quoted strings and therefore cannot serve as a delimiter.

If blanks are everywhere ignored, then tokens must be delimited by other tokens. Consider this example in Fortran:

DO10I = 1 7 E 3 . G T . X

We have separated the characters of the statement for clarity. It happens that the lexical analyzer cannot determine whether this statement begins with the keyword "DO" or the identifier "DO10I" until the "E" is scanned. A DO statement would have to have another digit or a comma before the "E" position. (This example draws upon the Fortran rule that a DO loop variable must be fixed-point. If, in some Fortran extension, a DO loop variable could be floating-point, then the lexical analyzer would have to scan to the first "." before determining that the statement is an assignment.) On the other hand, if one or more blanks were required as delimiters of tokens, but not permitted within a token, the statement would have to read

```
DO10I = 17E3 . GT . X
```

A DO statement would then have to carry a separating blank between the DO and the next token, e.g.,

```
DO 10I=17,1,25
```

Other blanks are unnecessary, since the remaining tokens can be distinguished.

There are several distinct language policies regarding the use of blanks and reserved words, as follows:

1. No policy, as in Algol 60. The problem of distinguishing keywords and separating tokens is left to an implementation. With no policy, a program may be untransportable without extensive editing.

2. Blanks must be inserted as needed as separators, and keywords are reserved. Many implementations follow this policy, for the sake of simplicity of the compiler.

3. Blanks must be inserted, and keywords are not reserved. (e.g., PL/I).

4. Blanks are ignored, ala Fortran, and keywords are reserved. (e.g., ANS Fortran subset).

5. Blanks are ignored, and keywords are not reserved. (e.g., ANS full Fortran).

One of these policies usually applies to a language to be implemented. Of these, policy 2 is the easiest. The analyzer can easily distinguish tokens in a left-to-right scan, and keywords can be distinguished from user identifiers through a table lookup. Unfortunately, this policy can be vexing to a compiler user. For example, PL/I contains over 100 keywords. It is unreasonable to expect every user to know every one of these keywords and never declare any of them as an identifier. A policy 2 implementation will also yield obscure syntax errors unless great care is taken to look for and properly report errors stemming from the use of keywords as identifiers.

Policy 3 is somewhat more difficult to implement, assuming the language is unambiguous under the policy. With inserted blanks, tokens are easily distinguished. However, the compiler may have to scan several tokens into the source string on some assumption regarding an identifier before finding that the assumption was right or wrong. If wrong, it must backtrack and try an alternative. Keywords are rarely used as identifiers, hence if the first choice is always "keyword" (if a choice exists), the amount of backtracking required will be small.

Policy 3 is considerably aided by a language policy of predeclared variables. Then the analyzer can be alert to any keywords that have been declared in its backtracking process. A keyword that has not been previously declared can be positively identified as such and not confused with an identifier.

Even this procedure breaks down if a variable declaration can begin with a declared identifier. Fortunately, most common programming languages require some keyword in a declaration preceding the declared identifier. Thus in Fortran, a declaration is triggered by one of the keywords COMMON, DIMENSION, REAL, INTEGER, etc. In PL/I, a declaration is triggered by the keyword DECLARE or DCL, and the declaration must appear first in a block. In Pascal, a declaration is triggered by one of the keywords TYPE or VAR.

Now consider an identifier DCL in PL/I declared in some block. The compiler will evidently have a problem when DCL is encountered at the head of some inner block. Is this a declaration or some statement beginning with DCL? For example,

```
BEGIN
    DCL  DCL FIXED BINARY;
         .
         .
         .
    BEGIN
      DCL I FIXED BINARY;  /* This DCL is a problem */
      DCL := I;     /* So is this DCL */
      DCL := I-1;     /* This DCL is not a problem */
         .
         .
         .
    END;
END;
```

Once the compiler is into the executable statements of a block, a declaration is no longer permissable, and DCL must be the user identifier. Other keywords can be distinguished by the keyword marking technique discussed above, based on the previously declared identifiers.

Policies 4 and 5 are still more difficult. The lexical analyzer must now make some decisions based on substrings of statements. Without certain other restrictions in the language, lexical analysis of a policy 5 language may be impossible, and the language may in fact be ambiguous—all this despite the existence of a perfectly reasonable and unambiguous high-level grammar for the language.

An example of a policy 5 language is Fortran. What makes Fortran translatable and unambiguous are other properties, as follows:

1. A statement is bounded in length and has a well-defined origin. These are defined by special record and continuation conventions (described above). A reasonable analyzer strategy is then to first bring a complete statement into a string buffer (the maximum size required is 1360 characters—20 maximum lines at 68 characters per line), so that the scanning can backtrack as needed without requiring the source file handler to back up.

2. Each statement either is a replacement statement of the form

<name> <optional index> = <expr>

or a control statement with a keyword prefix, e.g.,

<keyword> <remainder of control statement>

3. Names are limited to seven characters.

These help limit the number of possibilities in a statement. Unfortunately, ANS Fortran permits a variable declaration to appear anywhere in the program, and variables need not be declared. This means that any statement, including the last one, is potentially a statement beginning with a user identifier masquerading as a keyword.

Now consider the replacement statement. If the <optional index> is present, some Fortran implementations permit an arbitrary expression for the index, e.g.,

X(A+B*(17−S)) = Y

ANS Fortran permits only simple expressions of the form A*X± B as an array index. An expression index could continue through most of the 1360 characters of the statement buffer. Also the left part of a replacement has the same form as the left part of an IF statement:

IF(<expr>) <number>,<number>,<number>

or a FORMAT statement:

FORMAT(<format specification>)

An IF statement cannot be distinguished from an indexed replacement until the character past the closing right parenthesis is scanned.

A FORMAT can also create difficulties for a lexical analyzer. For example, suppose the strategy is to match left and right parentheses in an attempt to locate the closing right parenthesis of an indexed replacement, ignoring other context. This appears on the surface to be reasonable. Unfortunately, this strategy fails on certain Fortran statements, for example:

FORMAT(5H223)=, I5, 4X)

Such a scan will continue past the replacement symbol "=", and falter on the comma. The difficulty, of course, lies in the failure to fully parse a replacement syntax (the "5H223" doesn't belong as an index).

A backtracking strategy can deal with all the special problems of Fortran and other languages with difficult lexical policies. While backtracking, nothing much else can be done, since it is possible that any additional work must be undone later. However, it may be possible to build the abstract syntax tree for the statement, with no other immediate action. The initial parsing assumption is that the statement is a replacement. When it is apparent that this assumption is wrong, it may be possible to make some minor changes in the tree to reflect the alternative, and continue parsing. This approach requires that the parser be sufficiently general to accept both kinds of statement and that the parse trees for the alternatives are not too different. The symbol table should not be changed until the possibility of a wrong choice is eliminated.

Comments and Quoted Strings

A comment in Algol 60 begins with the reserved word COMMENT and ends on the next semicolon. Other languages have a similar arrangement. In Pascal, a comment is enclosed in braces { }, and in PL/I a comment is enclosed in the character pairs /* and */.

A quoted string is used as a data element in many languages, and is delimited by a pair of quotation marks. The beginning and ending quotation marks are sometimes the same character and sometimes different characters. In any case, the rule for quoted strings usually is that the string begins after the first quotation mark and continues to the terminating quotation, through whatever characters appear—blanks, comment strings, etc.

In dealing with comments and quoted strings, the lexical analyzer must have three states: an *outer* state S in which it looks for language tokens, a *quote* state Q in which it is scanning a quoted string, and a *comment* state C in which it is scanning a comment string. In state S, it may transfer to state Q on a quote mark or to state C on a comment opener. It remains in S for all other tokens. While in state C, the source is scanned and the analyzer should be sensitive to only two tokens—an end-of-file and a closing comment token. All others are ignored. The closing comment token causes a return to state S; an end-of-file should cause an error message and an orderly termination of the compile.

State Q is similar—only a closing quotation mark or an end-of-file is of interest.

Exercises

1. Outline a lexical analyzer for Algol 60. Start by making a list of the

tokens in the language, then examine the rules regarding reserved identifiers, blanks, and comments. Finally, design a finite-state automaton that will classify the tokens. Consider identifiers, quoted strings, and numbers as tokens.

2. Discuss the problem of efficient recognition of reserved words. For example, does it make sense to walk down through a tree on a sequence of letters, such that upon reaching some terminal node, the identifier ends? (You may wish to read chapter 6 next, or come back to this problem after studying chapter 6.)

3.8. Some FSA Theorems and Their Proofs

In this section, we prove a number of theorems stated earlier in the text.

3.8.1. Equivalence of Empty Cycle States

Let p and q be two states in a machine M such that $p \vdash^+ q$ and $q \vdash^+ p$ on empty moves only. Let x be any string accepted by M starting from state p (but p need not be the start state).

Then $p \vdash^+ q$ implies that there exists a sequence of states $p_1, p_2, \ldots p_n$ $(n \geq 0)$ such that

p_1 is in $\delta(p, \epsilon)$,
p_2 is in $\delta(p_1, \epsilon)$,
.
.
.
p_n is in $\delta(p_{n-1}, \epsilon)$, and
q is in $\delta(p_n, \epsilon)$

But then $(p, x) \vdash (p_1, x) \vdash (p_2, x) \vdash \ldots \vdash (p_n, x) \vdash (q,x)$ and therefore x is accepted by M starting from state q.

If we interchange "p" and "q" in this argument, we can show that any state x accepted by M in state q is also accepted by M in state p, therefore states p and q must be equivalent. QED.

3.8.2. Equivalence Through Removal of Empty Moves

A simple algorithm for the removal of empty transitions from a machine M to yield a machine M′ was given in section 3.2.2. We have already shown that the algorithm terminates (the absence of empty cycles is crucial). It should be clear that no empty moves remain. We need to show equivalence. We do this by proving the somewhat stronger lemma:

Lemma. For all (x in Σ^* , p in Q) ($(p, x) \vdash^* (f, \epsilon)$ in M if and only if $(p', x) \vdash^* (f', \epsilon)$ in M') where f is in F and f' is in F'.

For convenience, we indicate the states in M' with primed letters, e.g., p', q', and the states in M with unprimed letters, e.g., p, q. Q is the state set in M, and F is the set of halt states in M. The algorithm does not remove or add any states, so that every state p in M corresponds to a state p' in M'.

Proof: "Only-if" part. Let $(p, x) \vdash^* (f, \epsilon)$ in k moves, $k \geq 0$.

The basis step (k = 0). Clearly p = f and is in F. But p' must be in F' by state correspondence.

The inductive step (k > 0). Consider the first step in the machine move sequence:

$$(p, ax) \vdash (q, x) \vdash^* (f, \epsilon), \text{ where a is in } \Sigma \cup \{\epsilon\}$$

We have several cases to consider. If a is nonempty, then q' is in $\delta(p',a)$ and therefore $(p',ax) \vdash (q',x)$. But then (q',x) must yield (f, ϵ) in k-1 moves or less, by the inductive hypothesis, hence M' accepts ax in state p'.

If a is empty, we have $(p, x) \vdash (q, x) \vdash^* (f, \epsilon)$, with q in $\delta(p, \epsilon)$. But q' is not in $\delta(p', \epsilon)$ in M'. We now have two subcases to consider: x empty or not.

If x is empty, then $(p, \epsilon) \vdash (q, \epsilon) \vdash^* (f, \epsilon)$. The complete sequence involves only empty moves, with a sequence of states $(p, q, q_1, q_2, \ldots, f)$. By the halt state rule in the construction process, each of the states p', q', $q'_1, \ldots$ must be halt states. But then (p', ϵ) is an accepting configuration for M'.

If x is nonempty, there must be some first nonempty move in the move sequence, as follows:

$$(p, ax) \vdash (p_1, ax) \vdash \ldots \vdash (p_n, ax) \vdash (q, x) \vdash^* (f, \epsilon)$$

But by the construction process, we must have

q' in $\delta(p'_n, a)$,
p'_{n-2} in $\delta(q',a)$,
.
.
.
p' in $\delta(q',a)$.

Hence, $(p', ax) \vdash (q', x) \vdash^* (f', \epsilon)$. (The latter set of moves follows from the inductive hypothesis; there are less than k moves in that sequence). QED

A proof of the "if" part is similar in character.

3.8.3. Equivalence on the NDFSA to DFSA transformation

We show that L(M) is a subset of L(M'), where M' is derived from M by the NDFSA → DFSA transformation of section 3.2.3.

Proof: Let $(p, x) \vdash^* (f, \epsilon)$ in M in k steps. We show that $([\ldots p\ldots], x) \vdash^* ([\ldots f\ldots], \epsilon)$ in M', for every state $[\ldots p\ldots]$ in M'.

Basis, k = 0: Here p = f and f is in F. But then $[\ldots p\ldots]$ must be in F', hence x is accepted by M.

Inductive step, k > 0: Consider the first step

$$(p, ax) \vdash (q, x) \vdash^* (f, \epsilon)$$

where "a" is in the alphabet. (*Note*: No empty moves can exist). But then we have q in $\delta(p, a)$ and therefore $[\ldots q\ldots]$ is in $\delta([\ldots p\ldots], a)$ in M' for every state of the form $[\ldots p\ldots]$ in M'. Then

$$([\ldots p\ldots], ax) \vdash ([\ldots q\ldots], x) \vdash^* ([\ldots f\ldots], \epsilon)$$

The proof that L(M') is a subset of L(M) is similar. QED

3.8.4. The Pairs Table Reduction Algorithm

We divide the proof into a lemma and a theorem. The theorem applies to the pairs table upon completing its construction, and the lemma to the completion of the marking process.

Lemma. The feasible state-pairs contain all the equivalent state-pairs of the FSA.

Theorem. The unmarked state-pairs contain all and only the equivalent state-pairs of the FSA.

A proof of the lemma is left as an exercise. A proof of the theorem follows.

Proof: The "All" Part. Let (p, q) be an equivalent state-pair in the FSA. Then by the lemma, it is among the feasible state-pairs in the construction of the pairs table. Now suppose (p, q) becomes marked during the marking phase, and therefore is not among the unmarked set. It became marked because some token caused a transition from it to some marked or absent state-pair, (p', q'). If (p', q') were absent, then states p' and q' must be distinguishable, by the lemma, and therefore (p, q) are distinguishable states. This statement contradicts the assertion of their equivalence. Suppose instead that the state-pair (p', q') is marked. By a repetition of this argument, there must have

been a transition from it to some other state-pair that caused it to be marked. The first such marking in this chain had to be caused by a transition to a missing state-pair. Hence (p′, q′) is distinguishable, and (p, q) must be distinguishable, which again contradicts the assertion of equivalence.

Proof: The "Only" Part. Let (p, q) be an unmarked pair appearing in the pairs table at the end of the process, but p and q are distinguishable in the machine. That they appear in the feasible set is no evidence of equivalence or distinguishability. However, if they are distinguishable, then there is some string $x = a_1a_2a_3 \ldots a_n$ such that x is accepted by the machine in state p, but not in q, or vice versa. Without loss of generality, assume that q is the non-accepting state. Now the failure to accept can be the result of a machine block prior to completing string x, or the result of completing x, but terminating in a non-halt state. Suppose first that the machine blocks. Then the state sequences beginning with p and with q look like this:

$$p \vdash p_2 \vdash p_3 \vdash \ldots \vdash p_n \qquad \text{(acceptance)}$$

$$q \vdash q_2 \vdash \ldots \vdash q_m \qquad \text{(block)}$$

where $m < n$, and $n = |x|$. Now states p_m and q_m are clearly not in the feasible state-pair set, since at least one transition (on the mth token of string x) can occur on p_m, but not on q_m. It follows that (p_{m-1}, q_{m-1}) is either not in the feasible state-pair or has become marked, since there is a transition from the pair (p_{m-1}, q_{m-1}) to (p_m, q_m), and the latter is not in the feasible state-pair set. Similarly, (p_{m-2}, q_{m-2}) become marked, etc., and eventually (p, q) become marked.

If the failure to recognize string x in state q is because of a nonhalt state upon completing x, then the state sequences beginning with p and with q look like this:

$$p \vdash p_2 \vdash p_3 \vdash \ldots \vdash p_n$$

$$q \vdash q_2 \vdash q_3 \vdash \ldots \vdash q_n$$

Here, p_n is in the halt set, but q_n is not. The pair (p_n, q_n) therefore cannot be in the feasible state-pair set. Thus (p_{n-1}, q_{n-1}) become marked, and because they are marked and there is a transition step from (p_{n-2}, q_{n-2}) to (p_{n-1}, q_{n-1}), the pair (p_{n-2}, q_{n-2}) become marked, etc. Eventually (p, q) becomes marked. QED

3.9. Bibliographical Notes

The earliest work on FSA is in McCullough [1943]. Kleene [1952] and [1956] first introduced the notion of regular expressions. Algorithms for the

interconversion of state transition functions and regular expressions are found in McNaughton and Yamada (McNaughton [1960]). An early review paper is Brzozowski [1962]. The material on equivalence is largely from Huffman [1954], Moore [1956], Mealy [1955], and Aufenkamp and Hohn (Aufenkamp [1957]), as found in Gill [1962]. A regularity test for a context-free grammar is given by Stearns [1967]. The literature on FSA and their applications to logic circuitry is very large; some representative texts are Booth [1967], Gill [1962], Harrison [1965], Hartmanis and Stearns (Hartmanis [1966]), Kohavi [1971], Maley and Earle (Maley [1963]), McCluskey [1965a], and McCluskey and Bartee (McCluskey [1965b]). In addition, Aho and Ullman (Aho [1972a] and Hopcroft and Ullman (Hopcroft [1969]) contain material on finite-state automata and their relation to language recognition. Lexical analysis has been discussed by many authors; a representative sample is Johnson [1968], Conway [1963], DeRemer [1974c], Gries [1971], Feldman [1968].

CHAPTER 4

TOP-DOWN PARSING

A top-down parser conceptually develops a derivation tree for a sentence in the language from the root node down. We have seen previously that the essential problem is that of deciding which of several productions with the same left-member applies in the tree next. We always know the next left-member, for it is associated with a tree node known to be a part of the final derivation tree.

We first address ourselves to the general problem of a top-down translator for a context-free language by examining a general nondeterministic parser. Although a nondeterministic machine is impractical in a real compiler, it exhibits many of the properties that must exist in a real compiler. Furthermore, any context-free language can be parsed by such a machine and we can easily prove that the machine recognizes exactly the language of the context-free grammar used to construct it.

A deterministic top-down parsing machine unfortunately cannot be constructed for every context-free grammar; there are context-free languages which cannot be recognized by a deterministic top-down parsing machine.

However, those grammars with a deterministic top-down parser are sufficiently powerful to define many common programming languages and therefore to construct useful and efficient compilers.

4.1. Nondeterministic Push-Down Automata

The nondeterministic top-down parser we shall construct is an example of a general class of push-down automata, or PDA in short. A *push-down*, or *stack automaton*, contains (1) an *input string*, of symbols in the alphabet of the machine, (2) a *read head*, which may examine one symbol in the input list at a time and may move only from left to right, (3) a *finite-state machine*, which serves to control the system's operations, and (4) a *last-in-first-out push-down stack*. See figure 4.1(a).

A *move* in a PDA is governed by the present state, the input symbol under the read head, and the symbol on the top of the push-down stack. In a move, the read head is advanced, the state is changed, and the top stack symbol is replaced by some string (possibly empty).

A PDA scans its input string by a succession of such moves. It may be unable to move at some point, in which case some other set of choices made earlier must be tried. If a set of choices permits the machine to scan the input

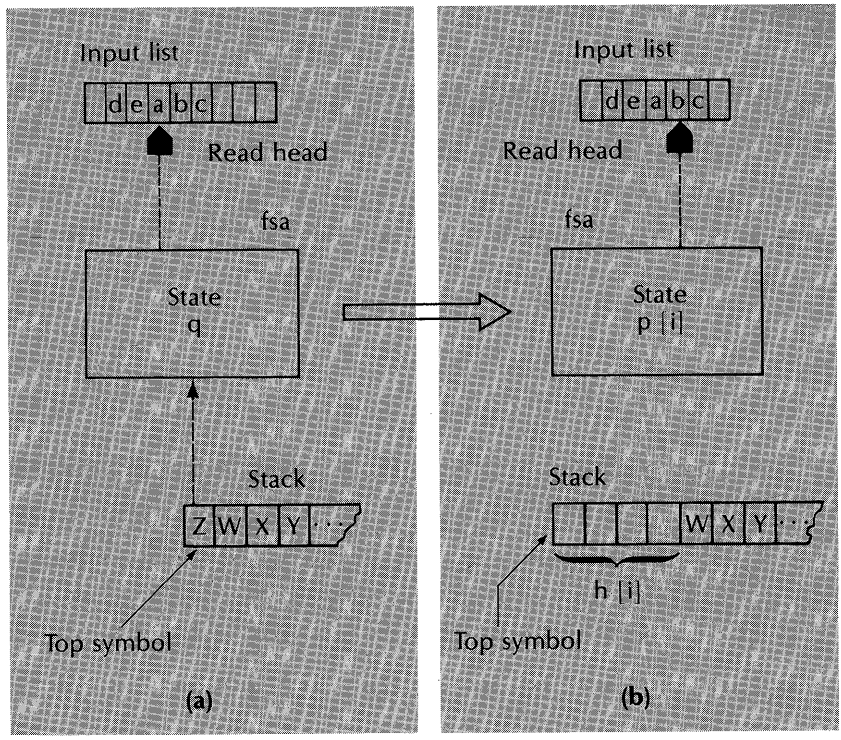


Figure 4.1. Non-empty move of a push-down automaton. The read head is advanced one token in the input list, the state changes from q to p[i], token Z on the stack is replaced by a string h[i].

string and either end in a halt state, or empty its stack, then the input string is said to be *accepted* by the PDA.

A PDA is therefore a 7-tuple (K, Σ, H, δ, q_0, Z_0, F), where:

- K is a finite set of states.
- Σ is a finite input alphabet, i.e., the set of symbols appearing in the input list.
- H is a finite push-down stack alphabet. For generality, we permit different symbols to be used on the push-down stack than elsewhere.
- $q_0 \in K$ is the initial state of the machine.
- $Z_0 \in H$ is the initial symbol on the push-down stack.
- $F \subset K$ is a set of final or halt states.
- δ is a transition function, mapping a triple (q, a, Z) to a set of pairs $\{(p_1, h_1), (p_2, h_2), \ldots\}$, where q is in the state set K, "a" is a member of $\Sigma \cup \{\epsilon\}$, Z is in the push-down stack alphabet H, $p_1, p_2, \ldots$ are members of the state set K, and $h_1, h_2, \ldots$ are strings in H^* .

A PDA move is governed by the transition function δ, just as in a finite-state automaton. However, in a push-down automaton, the move is controlled not only by the next input symbol and the present state but also by the symbol on the top of the push-down stack. Furthermore, the result of a move is not only some new state, but also the replacement of the top symbol on the push-down stack by a string of symbols drawn from the stack alphabet H.

Let us explore a typical move in more detail. Consider a member of the transition function:

$$\delta(q, a, Z) = \{(p_1, h_1), (p_2, h_2), \ldots (p_m, h_m)\} \tag{4.1}$$

In figure 4.1(a), the PDA is shown in a state in which symbol "a" is under the read head, the controlling FSA is in state q, and the top symbol on the push-down stack is Z. These are the conditions necessary to invoke the move expressed by the transition function member, Eq. (4.1) above. The move is made by choosing one of the pairs in the set $\delta(q, a, Z)$, assuming that at least one pair exists. Suppose we choose the pair (p_i, h_i). Then the next state of the FSA is p_i, the symbol Z on the stack top is replaced by the string h_i, and the read head is advanced to the next symbol in the input list, figure 4.1(b). (Note that the push-down stack is drawn with its top to the left, which is useful in describing a top-down PDA, as we shall see.)

There are two possible variations on Eq. (4.1). The symbol "a" in $\delta(q, a, Z)$ may be the empty string, ϵ. If this is so, then the machine may move without considering the input symbol; it may move on the basis of its state and the top stack symbol alone. The move is also made without moving the read head.

In a second variation, the string h_i in a transition set pair may be empty. Then the stack top symbol Z is effectively popped from the stack, exposing the symbol (if any) beneath it. With this variation, the stack can be reduced and ultimately emptied.

A PDA may halt in either of two ways—by empty stack or by final state. It is said to halt by empty stack and accept the input string if, upon the move in which the read head advances just past the end of the input list, the push-down stack is emptied. It is said to halt by final state and accept the input string if, upon the move in which the read head advances just past the end of the input list, the FSA enters a member of the final state set F. A given PDA is always defined to halt in one manner or the other. However, it can be shown that for every PDA of one kind, there exists an equivalent PDA of the other kind, so that we may choose whichever is more convenient.

In either case, a nondeterministic PDA is said to accept a given input string if there exists a sequence of moves that lead from its initial state and stack contents to a halt condition.

The PDA is said to *block* if it fails to accept an input string.

Configurations and moves

A *configuration* of a PDA is a triple (q, w, h), where q is some state in K, w is the portion of the input list from the read head position to its right-end and a member of Σ^*, and h is the contents of the push-down stack and a member of H^*.

A configuration contains all the information needed to predict the future behavior of a PDA. For example, the PDA of figure 4.1(a) may be described by the configuration

$$(q, abc\ldots, ZWXY\ldots)$$

The PDA of figure 4.1(b) may be described by the configuration

$$(p_i, bc\ldots, h_iWXY\ldots)$$

A move in a PDA may be regarded as a means of transforming one configuration into another one, which provides a more concise way of defining a machine move, as follows.

We say that

$$(q, ax, Zh') \vdash (p, x, hh')$$

is a possible move if and only if (p, h) is in $\delta(q, a, Z)$. (Note that this move definition is consistent with the possibility that $a = \epsilon$, and $h = \epsilon$.)

A sequence of one or more moves is denoted by $\vdash^+$. A sequence of zero or more moves, where a "zero" move is no change in the present configuration, is denoted by $\vdash^*$. With this notation, we may define the language L(M) recognized by a PDA M as:

$$L(M) = \{ w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon) \} \tag{4.2}$$

for some p in K, where machine M halts by empty stack, or as:

$$L(M) = \{ w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, h), p \text{ in } F \} \tag{4.3}$$

for some h in H^*, where machine M halts by final state.

The definition in (4.2) may be translated to English roughly as follows: The language L(M) of a PDA M consists of all those strings w such that, given a PDA initially in state q_0, with stack contents Z_0, and the read head positioned at the first symbol of w, there exists a sequence of moves that results in the read head completing the input string, an empty stack, and some end state (not necessarily a final state).

The definition in (4.3) may be similarly interpreted.

Example Consider the two-state machine defined by figure 4.2. The states are P and Q, the input symbols are 0 and 1, and the stack symbols are R, B, and G. This machine recognizes all palindromes in the symbols 0 and 1, i.e., the language

	"From"			"To"			
Row	State p	Input symbol a	Stack symbol z	State q_1	Stack h_1	State q_2	State h_2
1	P	0	R	P	BR		
2	P	1	R	P	GR		
3	P	0	B	P	BB	Q	ϵ
4	P	0	G	P	BG		
5	P	1	B	P	BB		
6	P	1	G	P	GG	Q	ϵ
7	P	ϵ	R	Q	ϵ		
8	Q	0	B	Q	ϵ		
9	Q	1	G	Q	ϵ		
10	Q	ϵ	R	Q	ϵ		

$$d(p, a, Z) = [(q_1, h_1), (q_2, h_2), \ldots]$$

Figure 4.2. A two-state push-down automaton, recognizing the strings $\{ww^R \mid w \in (0,1)^*\}$, the palindromes in $\{0,1\}^*$.

$$\{ w w^R \mid w \in \{0,1\}^* \}$$

where w^R is the string w, but reversed in order.

This machine is nondeterministic because the moves in rows 3 and 6 contain alternative moves, and also because a move may be made on no input symbol (row 7), and alternatives to this exist.

The starting state is P, and the initial stack contents is R. It halts by empty stack.

Consider the input string 001100, and let us trace the machine configurations:

(P, 001100, R)	$\vdash$	(P, 01100, BR)	by row 1
or	$\vdash$	(Q, 001100, ϵ)	by row 7 (block)
(P, 01100, BR)	$\vdash$	(P, 1100, BBR)	by row 3a
or	$\vdash$	(Q, 01100, R)	by row 3b
(Q, 01100, R)	$\vdash$	(Q, 01100, ϵ)	by row 10 (block)
(P, 1100, BBR)	$\vdash$	(P, 100, GBBR)	by row 5
(P, 100, GBBR)	$\vdash$	(P, 00, GGBBR)	by row 6a
or	$\vdash$	(Q, 00, BBR)	by row 6b
(P, 00, GGBBR)	$\vdash$	(P, 0, BGGBBR)	by row 4
(P, 0, BGGBBR)	$\vdash$	(P, ϵ, BBGGBBR)	by row 3a (block)
or	$\vdash$	(Q, ϵ, GGBBR)	by row 3b (block)

(Q, 00, BBR)	$\vdash$ (Q, 0, BR)	by row 8
(Q, 0, BR)	$\vdash$ (Q, ϵ, R)	by row 8
(Q, ϵ, R)	$\vdash$ (Q, ϵ, ϵ)	by row 10 (accept)

The above trace shows the consequences of running down the various blind alleys that occur; nevertheless, the machine ends in an acceptance of the string.

This PDA does its work essentially by pushing B on the stack for every 0 and G for every 1 found in the first half of the input string. It must nondeterministically decide when the second half of the string begins. If it makes the wrong choice for a turning point, it will either exhaust its stack too soon, or it will exhaust its input string before the stack is empty.

It remains in state P for the first half of the string, and switches to state Q for the second half. While in state Q, the input string elements are effectively matched against the stack symbols; 0 must correspond to a B on the stack, and 1 to a G.

The initial stack symbol R becomes buried in the stack as soon as some symbols in w are matched, and becomes uncovered only as the last symbol in the input string is matched. Upon uncovering R, the stack is also emptied, permitting a halt. R is never pushed onto the stack; it therefore serves as an end marker for the stack.

If the input string is not a palindrome, the machine is unable to find a sequence of accepting moves. It will fail through an inability to match the stacked B's and G's against the 0's and 1's in the second half of the input list in state Q, or it will fail through exhausting the stack before exhausting the input list or vice versa.

Exercises

1. Trace the PDA of figure 4.2 for the strings 010010, 1010 and 0111. Show that the latter strings cannot be accepted by any sequence of moves.

2. Design a PDA that recognizes palindromes in the alphabet {0, 1, C}, where C marks the center of the palindrome; e.g., 0110C0110 is a legal palindrome. A deterministic PDA for this language exists.

3. Design a PDA that accepts strings in {(,)} consisting of all correctly nested parentheses, e.g., (( ) ( )) is in the language, but ( )) ( ) (is not.

4. Define a backtracking system for a PDA. The tape need not consist of fixed-length cells. Can you find a PDA for which the backtracking system will fail? Can you characterize the class of PDA's for which your backtracking system will fail?

Context-Free Languages and PDA

An important theorem connecting PDA and context-free languages is the following:

Theorem 4.1. For every context-free language L there exists a nondeterministic PDA M such that M exactly recognizes L, and conversely.

The practical importance of this theorem in compiler systems lies in the realization that essentially the syntax of every programming language in existence can be expressed almost exactly by a context-free grammar. Then theorem 4.1 insists that a mechanical recognizer for the language must exactly be a push-down automaton.

Of course, every regular grammar is also context-free, by its very form. The converse is not true. There are context-free languages that are not regular. The palindrome language is an example of a nonregular language, yet we have seen that a PDA can recognize it. (The proof that a palindrome cannot be regular is beyond the scope of this book). It may also be possible to transform a grammar which appears to be context-free into a regular grammar. However, in general such a transformation is not always possible, and when it is not, we must have a recognizer with a push-down stack.

The push-down stack in a compiler may be explicitly coded into the program, or it may be hidden. A recursive-descent compiler is an example of a top-down context-free recognizer in which the PDA push-down stack is in fact a stack of return addresses formed during the procedure calls which constitute the parsing process. Since this return address stacking system is invisible to the user of a modern programming language, the parsing process appears not to involve a stack.

Theorem 4.1 has two faces. The most interesting one from the point of view of compiler construction is that a given context-free language can be recognized by some PDA M. The converse, that given a PDA M, one can construct an equivalent grammar from it, is also true, but hardly needed in compiler construction. We shall therefore define the machine construction and prove equivalence to the language of the given grammar.

Algorithm 4.1. CFG to a Nondeterministic PDA

The machine M will be nondeterministic, will have only one state, q, and will have the rules defined as follows:

- $\delta(q, \epsilon, A)$ contains (q, w) for every production $A \rightarrow w$ in P.
- $\delta(q, a, a)$ contains (q, ϵ) for every a in Σ.

The stack symbols are in $N \cup \Sigma$, and the stack initially contains the start symbol S. Machine M will accept by empty stack.

Example. Consider the simple grammar

$$S \rightarrow 0S1 \mid c$$

This grammar describes the language $\{0\}c\{1\}$, where the number of 0's and 1's are equal. The transition function rules are:

$$\begin{aligned}
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon) \\
\delta(q, c, c) &= (q, \epsilon) \\
\delta(q, \epsilon, S) &= \{(q, 0S1), (q, c)\}
\end{aligned}$$

A trace of the machine moves for the string 00c11 is given next. We omit the state from the configurations, since it is always the same.

(00c11, S) $\vdash$	(00c11, 0S1)	(O.K.)
or	(00c11, c)	(N.G.)
(00c11, 0S1) $\vdash$	(0c11, S1)	
(0c11, S1) $\vdash$	(0c11, 0S11)	
or	(0c11, c1)	(N.G.)
(0c11, 0S11) $\vdash$	(c11, S11)	
(c11, S11) $\vdash$	(c11, 0S111)	(N.G.)
or	(c11, c11)	
(c11, c11) $\vdash$	(11, 11)	
(11, 11) $\vdash$	(1, 1)	
(1, 1) $\vdash$	(ϵ, ϵ)	(acceptance)

Another Example. Empty production rules are acceptable, too:

$$S \rightarrow 0S1 \mid \epsilon$$

The transition function is

$$\begin{aligned}
\delta(q, 0, 0) &= (q, \epsilon) \\
\delta(q, 1, 1) &= (q, \epsilon) \\
\delta(q, \epsilon, S) &= \{(q, 0S1), (q, \epsilon)\}
\end{aligned}$$

Then the string 0011 is accepted as follows:

$$(0011, S) \vdash (0011, 0S1) \vdash (011, S1) \vdash (011, 0S11)$$

$$\vdash (11, S11) \vdash (11, 11) \vdash (1, 1) \vdash (\epsilon, \epsilon), \text{ acceptance.}$$

Discussion

By following the example machine traces just given, it may be seen that the effect of the rule

$$\delta(q, a, a) = (q, \epsilon) \qquad \text{for all } a \in \Sigma$$

is to *match* input terminal symbols against terminal symbols on the stack top. The move is only possible when the two symbols match, and the result is to move the read head and pop the symbol from the stack. This matching operation ceases only when a nonterminal symbol appears on the stack top. Then the other rule:

$$\delta(q, \epsilon, A) \quad \text{contains } (q, w) \text{ for every } A \rightarrow w \text{ in } P$$

applies. The machine must somehow choose among one of several productions with the same left member; the left member is known, since it is on the stack top.

Once a choice is made, the nonterminal A is replaced on the stack by the string w, which in general contains both terminals and nonterminals.

Recall that the production rule choice problem was also present in attempting to construct a derivation tree for a sentence from the top down. We have not solved the choice problem, but are examining it from a different point of view.

Exercise

Construct a PDA for grammar G_0, given below, then show that it accepts the string (a+a)*a but not the string (+a).

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Proof of Theorem 4.1: We prove the stronger result expressed by the next lemma.

Lemma 4.1. $(q, wx, Ay) \vdash^* (q, x, y)$ in M if and only if $A \Rightarrow^* w$ in grammar G, where A is in N, wx is in Σ^*, and y is in $(N \cup \Sigma)^*$.

This lemma includes theorem 4.1 as a special case, with A = S, and $x = y = \epsilon$.

Proof: If Part. We prove the if part of the lemma by induction on the number of steps in the derivation $A \Rightarrow^* w$. Consider a derivation of one step. Then $A \rightarrow w$ is a production, with w a terminal string. This production implies that there is a transition

$$\delta(q, \epsilon, A) \text{ contains } (q, w)$$

which yields the machine move:

$$(q, wx, Ay) \vdash (q, wx, wy)$$

Now w is a terminal string. If its length $|w| = n$, then we may invoke n applications of the matching rules, which have the form:

$$\delta(q, a, a) = \{(q, \epsilon)\}$$

for every terminal "a" in the alphabet. These n applications yield the configuration

$$(q, x, y)$$

which was to be shown.

Now assume that the derivation $A \Rightarrow^* w$ requires k steps, where $k > 1$, and that lemma 4.1 holds for all $k' < k$. The first step of the derivation has the form

$$A \Rightarrow X_1X_2X_3 \ldots X_n \Rightarrow^* x_1x_2x_3 \ldots x_n = w,$$

where $X_1 \Rightarrow^* x_1$, $X_2 \Rightarrow^* x_2$, etc., and $X_1, \ldots, X_n$ are in $N \cup \Sigma$.

Thus the move

$$(q, wx, Ay) \vdash (q, x_1x_2 \ldots x_nx, X_1X_2 \ldots X_ny)$$

exists. Now if X_1 is a terminal symbol, it must be equal to x_1, and the transition $\delta(q, x_1, x_1) = \{(q, \epsilon)\}$ may be made. If X_1 is nonterminal, then by the inductive hypothesis (since $X_1 \Rightarrow^* x_1$ by less than k moves), the moves

$$(q, x_1r, X_1s) \vdash^* (q, r, s)$$

for any strings r and s exist.

Thus in either case, the following moves exist:

$$(q, x_1x_2 \ldots x_nx, X_1X_2 \ldots X_n\ y) \vdash^* (q, x_2 \ldots x_nx, X_2 \ldots X_ny)$$

More repetitions of this process, for $X_2, \ldots, X_n$ eventually yields the configuration (q, x, y), which was to be shown. QED

Proof: Only If part. Let $(q, wx, Ay) \vdash (q, x, y)$ in k moves; we prove that $A \Rightarrow^* w$ by induction on the number of moves, k.

For $k = 1$, w must be ϵ and $A \rightarrow \epsilon$ is in P. (There are no other possibilities with A on the top of the stack, in one machine move). Thus assume the lemma valid for all moves of length $k' < k$; we prove it valid for k. The first move must have the form

$$(q, wx, Ay) \vdash (q, wx, X_1 \ldots X_ny)$$

where $A \rightarrow X_1 \ldots X_n$ is in P, and

$$(q, x_i, X_i) \vdash^* (q, \epsilon, \epsilon)$$

for all $1 \leq i \leq n$ in k' moves or less, where $w = x_1x_2 \ldots x_n$.

Then $X_i \Rightarrow^* x_i$ for all i, by the inductive hypothesis; but putting all this together, we find

$$A \Rightarrow X_1 \ldots X_n \Rightarrow^* x_1 X_2 \ldots X_n \Rightarrow^* x_1 x_2 X_3 \ldots X_n \Rightarrow^* \ldots \Rightarrow^* x_1 x_2 \ldots x_n = w$$

QED

The Input List, the Stack, and Sentential Forms

An interesting property of the NDPDA defined in algorithm 4.1 is expressed by the following theorem:

Theorem 4.2. Let (q, y, h) be any configuration of the NDPDA for some grammar G, where the input string is xy, such that

$$(q, xy, S) \vdash^* (q, y, h)$$

Then xh is a left-most sentential form in G, i.e., $S \Rightarrow^* xh$.

The converse is also true: Given any sentential form xh in G, such that x is a terminal string, and at most one left-most symbol in h is terminal, then (q, y, h) is a configuration of the NDPDA such that

$$(q, xy, S) \vdash^* (q, y, h)$$

To visualize the significance of this theorem, consider figure 4.1(a). The string x is ". . .de", y is "abc..", and the stack string h is "ZWXY..". Then the theorem says that ". . .deZWXY.." is a sentential form. In short, the input list to the left of the read head, when concatenated with the stack, is a sentential form.

To prove this, note that it is trivially true for the initial configuration; the input string prefix x is empty, and S is on the stack. Therefore, assume that the theorem holds for n moves of the NDPDA, $n \geq 0$, and consider the $(n+1)$th move. This move either is a *matching* move, or a production rule *replacement* move. If a matching move, the string xh is clearly unchanged, since first symbol in h effectively is moved to the last position in x.

If a replacement move, the string xh before the move is of the form xAz, where $h = Az$. After the move, the string has the form xwz, where $A \rightarrow w$ is a production. By the inductive hypothesis, xAz is a sentential form, i.e.,

$$S \Rightarrow^* xAz$$

But since $A \rightarrow w$ is a production, xwz is also a sentential form:

$$S \Rightarrow^* xAz \Rightarrow xwz$$

QED

A proof of the converse is similar.

4.2. LL(k) Grammars

The LL(k) grammars are a proper subset of the context-free grammars. They are the largest such class that permits deterministic left-to-right top-down recognition with a lookahead of k symbols.

The deterministic top-down parsing problem is represented by figure 2.11, in chapter 2. We have some nonterminal node (T in the figure), and an uncompleted string, "(a*a)" in the figure. If the correct production can be deduced from the partially constructed tree and the next k symbols in the unscanned string, for every possible top-down parsing step, then we say that the grammar is LL(k). It is usually not obvious whether a given grammar is LL(k), nor can we ordinarily examine every possible parsing step. We seem to need two algorithms: one to test a grammar for the LL(k) condition and another one to generate a parsing table, similar to figure 2.16, for some grammar. It turns out that only the latter algorithm is needed; the failure to generate a parsing table will be apparent from the algorithm, and this failure will mean that the grammar is not LL(k). If the table generation succeeds, then the grammar is LL(k), and we will not only have a definition of a deterministic parser, but will also know that the grammar is unambiguous.

4.2.1. Definitions

The discussion on lookahead of k symbols is considerably simplified if there always exist at least k symbols to examine. For this reason, we introduce a new terminal symbol $\perp$ not already in the grammar, and append k of these symbols to every input string, prior to parsing. Hence we introduce a new nonterminal S′ and a production

$$S' \rightarrow S \perp^k$$

where S is the grammar's start symbol, and $\perp^k$ is a string of k symbols $\perp$. The symbol S′ becomes the new start symbol.

Definition 1. A production $A \rightarrow x_1$ in a context-free grammar G is called an *LL(k) production*, if in G,

$$S' \Rightarrow^* w\, A\, y \Rightarrow w\, x_1\, y \Rightarrow^* w\, z\ldots, \text{ and}$$

$$S' \Rightarrow^* w\, A\, y' \Rightarrow w\, x_2\, y' \Rightarrow^* w\, z\ldots$$

with $|z| = k$ implies $x_1 = x_2$.

Definition 2. A nonterminal in a context-free grammar is called an *LL(k) nonterminal* if all its productions are LL(k) productions.

Example. In the grammar

$$S \rightarrow Abc \mid aAcb$$

$$A \rightarrow \epsilon \mid b \mid c$$

S is an LL(1) nonterminal, while A is an LL(2) nonterminal. The strings derivable from S are (first production) bc, bbc, cbc, and (second production) acb, abcb, accb. Clearly, the second production is uniquely selected on the lookahead symbol "a", while the first is selected on symbols "b" or "c".

For the A productions, we need to consider the sentential forms in which A is embedded, e.g., "Abc" and "aAcb". By the definition, the string preceding A is different, so we need only consider whether the three A productions can be distinguished within "Abc" and within "aAcb". In the former, we have the three lookahead strings "bc", "bb" and "cb"; these are distinguishable with k = 2 but not with k = 1. In the latter production, the lookahead strings are "cb", "bc", and "cc", again distinguishable with k = 2 but not k = 1. The A productions are therefore LL(2), but not LL(1).

Note that if the lookahead sets were considered without regard to the particular sentential form prefix w in $S \Rightarrow^* wAy$, then the A nonterminal requires three lookahead symbols.

It is clear that an LL(k) grammar is also an LL(k−1) grammar, for $k \geq 1$.

4.2.2. Some Properties

Theorem 4.2.1. Each LL(k) grammar is unambiguous.

An ambiguity would lead to a contradiction with definitions 1 and 2.

Theorem 4.2.2. An LL(k) grammar has no left-recursive nonterminals, i.e., nonterminals A such that $A \Rightarrow^+ Aw$ for some w.

Proof: Suppose that a nonterminal A_0 is left-recursive. Then there is some sequence of nonterminals $A_0, A_1, \ldots, A_n$ that are all left-recursive, and such that

$$A_0 \Rightarrow^+ A_1x_1 \Rightarrow^+ A_2x_2 \Rightarrow^+ \ldots \Rightarrow^+ A_nx_n$$

and $A_n = A_0$. Now at least one of these must have another production, otherwise all of them would be useless and could be deleted from the grammar. Also, at least one of the x_i must be incapable of deriving the empty string; if they all could, then we could have a derivation sequence $A_0 \Rightarrow^+ A_0$,

and an obvious ambiguity. Thus one of the A_i (call it A from here on) is such that

$$A \Rightarrow Bx' \Rightarrow^+ Ax$$

where $|x| > 0$, and also $A \rightarrow y$, where y is not Bx'. Then we can construct the derivation sequences

$$S \Rightarrow^+ rAx^k \ldots \Rightarrow rBx'x^k \ldots \Rightarrow rAx^{k+1} \ldots \Rightarrow ryx^{k+1} \ldots \Rightarrow^* rz \ldots$$

$$S \Rightarrow^+ rAx^k \ldots \Rightarrow ryx^k \ldots \Rightarrow^* rz \ldots$$

where $|z| = k$, and they contradict the assumption that the grammar is LL(k). QED

The LL(k) definition states that, given a left-most sentential form wAx, such that w matches the first |w| symbols of the input string, then the next production $A \rightarrow y$ can be inferred from the next k input symbols. It appears that in order to select the production $A \rightarrow y$ we need a mapping of all strings wAx' to the production set, where $|x'| = k$. The resulting LL(k) parsing table would be impossibly large. The following two theorems show that we do not need such a complete mapping—a mapping of strings Ax' to the production set is sufficient. Before we can introduce these theorems, we need to develop two useful functions $FIRST_k(w)$ and $FOLLOW_k(w)$.

FIRST and FOLLOW sets

The domain of $FIRST_k$ is some string w in $(N \cup \Sigma)^*$ and the domain of $FOLLOW_k$ is a nonterminal A in N. The functions are then defined as follows:

$$FIRST_k(w) = \{ x \mid w \Rightarrow^* xy, xy \in \Sigma^* , \text{ and}$$

$$(\text{if } |xy| < k \text{ then } y = \epsilon \text{ else } |x| = k)\}$$

$$FOLLOW_k(A) = \{ x \mid S \Rightarrow^* uAy \text{ and } x \in FIRST_k(y)\}$$

where the derivations are left-most. That is, $FIRST_k(w)$ for some string w is the set of all leading terminal strings of length k or less in the strings derivable from w. A string x in $FIRST_k(w)$ is less than k in length only if x is fully derivable from w, i.e., $w \Rightarrow^* x$, and $|x| < k$. The empty string is in $FIRST_k(w)$ if $w \Rightarrow^* \epsilon$. Note that w may include or consist of nonterminals.

$FOLLOW_k(A)$ is the set of all derivable terminal strings of length k or less that can follow A in some left-most sentential form. The empty string ϵ will be in $FOLLOW_k(A)$ if A is the last symbol in some sentential form. In particular, ϵ is in $FOLLOW_k(S)$, where S is the start symbol.

With these definitions, the following useful properties can readily be proven. In these, the range of the $FIRST_k$ set is extended to include a set of strings; i.e., $FIRST_k(U\ V)$, where U and V are sets of strings, is the union of all $FIRST_k(u\ v)$, where u is in U and v is in V. Also $FIRST_k(w) = \epsilon$ for $k \leq 0$.

1. $FIRST_k(aw) = a\ FIRST_{k-1}(w)$ for any string w, where a is in Σ.
2. $FIRST_k(\epsilon) = \{\epsilon\}$.
3. $FIRST_k(x\ y) = \overline{FIRST}_k(\overline{FIRST}_k(x)\ \overline{FIRST}_k(y)) = FIRST_k(x\ FIRST_k(y)) = FIRST_k(FIRST_k(x)\ y)$.
4. Given a production $A \rightarrow w$ in G, $FIRST_k(A)$ contains $FIRST_k(w)$.
5. Given a production $A \rightarrow xXy$ in G, $FOLLOW_k(X)$ contains $FIRST_k(y\ FOLLOW_k(A))$. Note that y may be the empty string.
6. $FOLLOW_k(S)$ contains ϵ, where S is the start symbol of G.

Properties 1 and 2 should be obvious from the definitions. Property 3 expresses associativity of the $FIRST_k$ operator and concatenation; however, note that $FIRST_k(x\ y)$ is not identical to $FIRST_k(x)\ FIRST_k(y)$, since the former may be of length k at most and the latter may be of length greater than k.

Property 4 follows immediately from the definition.

Example. Consider the following simple grammar G_1. Let us compute $FIRST_k$ and $FOLLOW_k$ for its nonterminals, for k = 1:

1. $G \rightarrow E\ \perp$
2. $E \rightarrow TE'$
3. $E' \rightarrow +E$
4. $E' \rightarrow \epsilon$
5. $T \rightarrow FT'$
6. $T' \rightarrow *T$
7. $T' \rightarrow \epsilon$
8. $F \rightarrow (E)$
9. $F \rightarrow a$

- $FIRST_1(F) = \{ (, a\}$ from productions 8 and 9
- $FIRST_1(T') = \{*, \epsilon\}$ from productions 6 and 7
- $FIRST_1(T) = FIRST_1(FT') = FIRST_1(F\ FIRST_1(T')) = \{ (, a\}$
- $FIRST_1(E') = \{ +, \epsilon\}$ from productions 3 and 4
- $FIRST_1(E) = FIRST_1(TE') = \{ (, a\}$
- $FIRST_1(G) = FIRST_1(E) = \{ (, a\}$

- $\text{FOLLOW}_1(E) = \{\perp,)\} \cup \text{FOLLOW}_1(E')$ from productions 1, 3, and 8
- $\text{FOLLOW}_1(G) = \{\epsilon\}$ using property (6)
- $\text{FOLLOW}_1(E') = \text{FOLLOW}_1(E)$ from production 2
- $\text{FOLLOW}_1(T) = \text{FIRST}_1(E' \text{FOLLOW}_1(E)) \cup \text{FOLLOW}_1(T')$ from productions 2 and 6
- $\text{FOLLOW}_1(T') = \text{FOLLOW}_1(T)$ from production 5
- $\text{FOLLOW}_1(F) = \text{FIRST}_1(T' \text{FOLLOW}_1(T))$ from production 5

By using these relations repeatedly, we obtain the following table of FOLLOW_1 sets:

	1	2	3	4	5	6
G	ϵ					
E	$\perp$	)				
E′			$\perp$	)		
T			$\perp$	)	+	
T′			$\perp$	)	+	
F			$\perp$	)	+	*

Columns 1 and 2 are the contents of $\text{FOLLOW}_1(G)$ and $\text{FOLLOW}_1(E)$ known directly from the productions. The remaining columns are determined by inference from these and the FOLLOW_1 and FIRST_1 relations given above.

Exercises

1. A grammar and its FIRST_1 and FOLLOW_1 sets are given below. Verify the grammar's FIRST and FOLLOW sets:

   $S \rightarrow SA \mid Ba$
   $A \rightarrow Ab \mid B \mid \epsilon$
   $B \rightarrow aA \mid c$

nonterminal	FIRST_1	FOLLOW_1
S	a, c	ϵ, a, b, c
A	ϵ, a, b, c	ϵ, a, b
B	a, c	ϵ, a, b

2. Prove the six properties from the definitions.

3. Develop an algorithm based on the six properties that yields the FIRST_k and FOLLOW_k sets for any grammar.

Some Implications of the LL(k) Definition

We now use the FIRST and FOLLOW functions in a set of theorems that will lead to an efficient LL(1) parser.

Theorem 4.2.3. Let G be a context-free grammar. Then G is LL(k) if and only if: For every distinct pair of productions $A \rightarrow u$ and $A \rightarrow v$, and every left-most sentential form wAx derivable in G, $\text{FIRST}_k(ux) \cap \text{FIRST}_k(vx) = \varnothing$.

The proof is left to an exercise.

Now suppose that G has no empty productions and that G is LL(1). Then neither u nor v are empty, nor can either of these derive ϵ. Clearly x doesn't matter, since $\text{FIRST}_1(ux) = \text{FIRST}_1(u)$ and $\text{FIRST}_1(vx) = \text{FIRST}_1(v)$. It also means that the particular sentential form in which A appears is no longer important. We then have the weaker condition expressed in the following theorem.

Theorem 4.2.4. Let G be a context-free grammar with no empty productions. Then G is LL(1) if and only if, for every pair of productions $A \rightarrow u$ and $A \rightarrow v$, $\text{FIRST}_1(u) \cap \text{FIRST}_1(v) = \varnothing$.

The proof is trivial, given Theorem 4.2.3.

Is there a similar LL(1) condition for grammars with empty productions? Consider Theorem 4.2.3 again. If G contains empty productions, then u or v can be empty, or can derive empty strings. If $u \rightarrow^* \epsilon$ is possible, then clearly $\text{FIRST}_1(ux)$ includes $\text{FIRST}_1(x)$, and $\text{FIRST}_1(x)$ need not be a subset of $\text{FIRST}_1(u)$. However, note that x follows A in the left-most sentential form wAx of the theorem, and this suggests the following theorem:

Theorem 4.2.5. Let G be a context-free grammar. Then G is LL(1) if and only if, for every pair of productions $A \rightarrow u$ and $A \rightarrow v$ the following condition holds:

$$\text{FIRST}_1(u\ \text{FOLLOW}_1(A)) \cap \text{FIRST}_1(v\ \text{FOLLOW}_1(A)) = \varnothing$$

The FIRST-FOLLOW condition is certainly necessary for G to be LL(1). If u can derive ϵ, then $\text{FIRST}_1(ux)$ contains $\text{FIRST}_1(x)$ and $\text{FIRST}_1(x)$ is a subset of $\text{FOLLOW}_1(A)$. We must show that the condition is also sufficient—which is perhaps surprising, since $\text{FOLLOW}_1(A)$ is not related to any one sentential form, but rather the entire class of sentential forms containing A. We therefore prove sufficiency—the "if" part of the theorem.

Proof: Suppose that G were not LL(1). Then there exists a pair of derivations

$$S' \Rightarrow^* wAx \Rightarrow wux \Rightarrow^* wz$$

$$\text{and } S' \Rightarrow^* wAx \Rightarrow wvx \Rightarrow^* wz'$$

where $u \neq v$, and $\text{FIRST}_1(z) = \text{FIRST}_1(z')$. Now

$$ux \Rightarrow^* z$$

$$\text{and } vx \Rightarrow^* z'$$

Now let $z = u'x'$ and $z' = v'x''$ where $u \Rightarrow^* u'$, $x \Rightarrow^* x'$, $v \Rightarrow^* v'$, and $x \Rightarrow^* x''$. Note that $\text{FIRST}_1(x')$ is in $\text{FOLLOW}_1(A)$, and $\text{FIRST}_1(x'')$ is also in $\text{FOLLOW}_1(A)$. We now examine three cases, depending on whether u' or v' or neither or both, are empty:

CASE 1

$u' \neq \epsilon$ and $v' \neq \epsilon$. Then clearly

$$\text{FIRST}_1(z) \subset \text{FIRST}_1(u) \subset \text{FIRST}_1(u\ \text{FOLLOW}(A))$$

$$\text{and } \text{FIRST}_1(z') \subset \text{FIRST}_1(v) \subset \text{FIRST}_1(v\ \text{FOLLOW}(A))$$

Then

$$\text{FIRST}_1(u) \cap \text{FIRST}_1(v) \neq \varnothing$$

QED
(We have proven that premise P implies Q by proving that $\sim$Q implies $\sim$P).

CASE 2

$u' = \epsilon$ and $v' \neq \epsilon$. Here,

$$\text{FIRST}_1(z') \subset \text{FIRST}_1(v) \cap \text{FIRST}_1(v\ \text{FOLLOW}(A))$$

$$\text{and } \text{FIRST}_1(z) = \text{FIRST}_1(x') \subset \text{FOLLOW}_1(A) \subset \text{FIRST}_1(u\ \text{FOLLOW}(A))$$

Again,

$$\text{FIRST}_1(u\ \text{FOLLOW}(A)) \subset \text{FIRST}_1(v\ \text{FOLLOW}(A)) \neq \varnothing$$

CASE 3

$u' = \epsilon$, $v' = \epsilon$. Here,

$$\text{FIRST}_1(z) = \text{FIRST}_1(x') \subset \text{FOLLOW}_1(A) \subset \text{FIRST}_1(u\ \text{FOLLOW}(A))$$

$$\text{and } \text{FIRST}_1(z') = \text{FIRST}_1(x'') \subset \text{FOLLOW}_1(A) \subset \text{FIRST}_1(v\ \text{FOLLOW}(A))$$

QED

Theorem 4.2.5 suggests the following generalization:

G is LL(k) if and only if for every pair of productions $A \rightarrow u$ and $A \rightarrow v$, $FIRST_k(u\ FOLLOW_k(A)) \cap FIRST_k(v\ FOLLOW_k(A)) = \varnothing$.

Unfortunately, this statement is not true for $k>1$. A grammar satisfying the above condition is said to be *strong LL(k)*, and it happens that not all strong LL(k) grammars are LL(k), as the example given earlier shows:

Let G have the productions

$$S \rightarrow Abc \mid aAcb$$
$$A \rightarrow \epsilon \mid b \mid c$$

Now $FOLLOW_2(A) = \{bc, cb\}$. Then for $A \rightarrow \epsilon$ we have

$$FIRST_2(\epsilon\ FOLLOW_2(A)) = \{bc, cb\}$$

For $A \rightarrow b$ we have $FIRST_2(b\ FOLLOW_2(A)) = \{bb, bc\}$. Hence this is not strong LL(2), yet it is LL(2), as we have shown previously.

Essentially, the set $FIRST_k(u\ FOLLOW_k(A))$ for a production $A \rightarrow u$ contains all of the k-symbol lookaheads derivable from Ax in all the left-most sentential forms wAx. However, a top-down decision step in an LL(k) parser is based on a particular left-most sentential form wAx, and the possible k-symbol lookaheads derivable from Ax is a subset of $FIRST_k(u\ FOLLOW_k(A))$, for a production $A \rightarrow u$.

The case $k>1$ is different from the case $k = 1$ essentially because a pair of symbols comprising a lookahead string can stem from a combination of derivations based on A and on the strings following A, and the combination causes certain grammars to be strong LL(k) but not LL(k).

4.3. Deterministic LL(1) Parser

The push-down automaton given in section 4.1 is nondeterministic for only one reason—when a nonterminal symbol A appears on the top of the stack, then a choice must be made among the set of productions $A \rightarrow w_1 \mid w_2 \mid w_3 \mid \ldots$ in P.

We therefore see that to make this parser deterministic, we need a table to select one of several productions. Such a table will consider the stack top symbol A and the next k input symbols in the input list; based on this pair, it must uniquely select a production $A \rightarrow w_i$.

An *LL(k) selector table* maps a pair (X,u) to a production $X \rightarrow w$, where X is a nonterminal symbol (on the stack top), and u is some terminal string, of length k. Such a table must yield a many-to-one mapping, since many possible strings u may correspond to a given production, yet a production

must be uniquely selected from the pair (X,u). The table size is finite, since there are a finite number of nonterminals and terminal strings of fixed length k.

Theorem 4.2.5 provides the justification for such a table, and also states that the grammars that can be so parsed are the strong LL(k) grammars.

4.3.1. LL(1) Selector Table

A finite selector table can be constructed for any k. However, its size grows very rapidly with k for a typical grammar. It turns out that in practice, if a grammar is not LL(1), it will likely not be LL(k) for any k. It is better to transform the grammar in an attempt to find an LL(1) grammar, rather than attempt to construct a parser with $k > 1$. We therefore restrict the following discussion to $k = 1$.

Note that a LL(0) grammar can contain at most one production $A \rightarrow w$ for each nonterminal A, and is therefore useless for any practical purpose–at most one terminal string can be derived in the grammar.

Before we describe the construction of a selector table, let us examine a typical LL(1) selector table, figure 4.3. This table describes a grammar G_1 whose language is the class of arithmetic expressions with all four operations $+ \; - \; * \; /$ and parenthesizing. The productions for the grammar are given in the right-hand column of figure 4.3. The language of this grammar is essentially the arithmetic language $L(G_0)$, chapter 2, however, G_0 is not LL(k) for any k, since it contains left-recursive productions, while G_1 is LL(1). Grammar G_0 can be transformed into G_1 by means to be described later.

LL(1) Parser Algorithm

1. Initially, the parser PDA contains the start symbol S on its stack top.
2. If the stack top contains a terminal symbol "a", then the input symbol must be an "a", else ERROR. If the two match, then advance the read head and pop the terminal symbol from the stack.
3. If the stack top contains a nonterminal symbol A, then examine the input symbol currently under the read head, and consult the LL(1) selector table (figure 4.3, for example) to determine which production is to be applied. If the selector table indicates production $A \rightarrow w$, then remove A from the stack and push the string w onto the stack.
4. The machine halts by empty stack.

Let us try this machine on the string "a – a – a". The initial configuration is therefore:

Non-terminal		Input	Production
1	E	a, (	$E \rightarrow TE''$
2	E″	+, -	$E'' \rightarrow T'E''$
3	E″	), ϵ	$E'' \rightarrow \epsilon$
4	T′	+	$T' \rightarrow +T$
5	T′	-	$T' \rightarrow -T$
6	T	a, (	$T \rightarrow FT''$
7	T″	*, /	$T'' \rightarrow F'T''$
8	T″	+, -,), ϵ	$T'' \rightarrow \epsilon$
9	F′	*	$F' \rightarrow *F$
10	F′	/	$F' \rightarrow /F$
11	F	a	$F \rightarrow a$
12	F	(	$F \rightarrow (E)$

Figure 4.3. An LL(1) selector table, for grammar G_1.

$$[a-a-a, E]$$

since E is the start symbol of the grammar. The selector table row 1 indicates that with E on the stack top and "a" the next symbol, $E \rightarrow TE''$ should be applied; TE'' replaces E on the stack top, yielding the configuration

$$[a-a-a, TE'']$$

Next, the selector table indicates that T should be replaced by FT'' on the stack top, yielding

$$[a-a-a, FT''E'']$$

Then the selector table indicates replacing F by a, yielding

$$[a-a-a, aT''E'']$$

At this point, the matching rule (2) applies, yielding the configuration

$$[-a-a, T''E'']$$

The remaining moves are

$$[-a-a, T''E''] \vdash [-a-a, E'']$$

$$\vdash [-a-a, T'E''] \vdash [-a-a, -TE'']$$

$$\vdash [a-a, TE''] \vdash [a-a, FT''E'']$$

$$\vdash [a-a, aT''E''] \vdash [-a, T''E'']$$

$$\vdash [-a, E''] \vdash [-a, T'E''] \vdash [-a, -TE''] \vdash [a, TE'']$$

$$\vdash [a, FT''E''] \vdash [a, aT''E''] \vdash [\epsilon, T''E''] \vdash [\epsilon, E'']$$

$$\vdash [\epsilon, \epsilon]$$

The derivation tree for this sentence and grammar is shown in figure 4.4. A comparison of the machine moves with this tree reveals that the machine effectively constructs the tree top-down by a left-to-right tree scan. That is, the derivation is

$$E \Rightarrow TE'' \Rightarrow FT''E'' \Rightarrow aT''E'' \ldots$$

for the first few steps.

Selector Table Construction

Most of our work is done. Given an algorithm for the FIRST and FOLLOW sets, an LL(1) selector table is very easy to build:

Let $A \rightarrow w$ be any production in the grammar, and let the terminal symbol x be in the set $FIRST_1(w\ FOLLOW_1(A))$. Then the selector table pair (A, x) maps to production $A \rightarrow w$.

If any pair (A,x) maps to two or more different productions, then the grammar cannot be LL(1); we say that we have a conflict.

For example, consider the grammar G_1 given previously. The selector table for this grammar is given next and is seen to be free of conflicts:

Pair	Production
E, (	$E \rightarrow TE'$
E, a	$E \rightarrow TE'$
E′, +	$E' \rightarrow +E$
E′,)	$E' \rightarrow \epsilon$
E′, ⊥	$E' \rightarrow \epsilon$
T, (	$T \rightarrow FT'$
T, a	$T \rightarrow FT'$
T′, *	$T' \rightarrow *T$
T′, +	$T' \rightarrow \epsilon$
T′, ⊥	$T' \rightarrow \epsilon$
T′,)	$T' \rightarrow \epsilon$
F, (	$F \rightarrow (E)$
F, a	$F \rightarrow a$
G, (	$G \rightarrow E\perp$
G, a	$G \rightarrow E\perp$

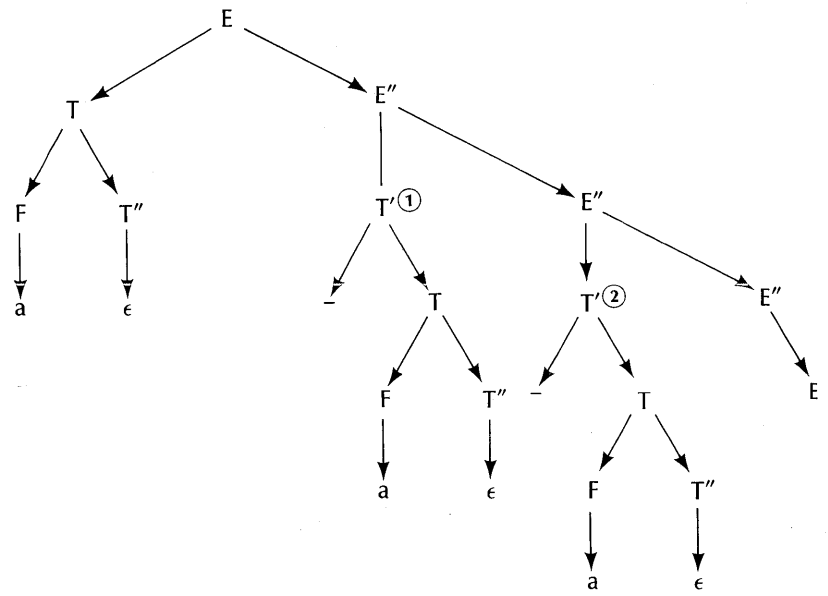


Figure 4.4. Derivation tree for string "a – a – a" in grammar G_1.

On the other hand, the following simple grammar has a selector table containing conflicts; it is therefore not LL(1):

$$S \rightarrow A\perp$$
$$A \rightarrow B \mid C$$
$$B \rightarrow aC$$
$$C \rightarrow aa$$

Pair	Production
S, a	$S \rightarrow A\perp$
A, a	$A \rightarrow B$
A, a	$A \rightarrow C$
B, a	$B \rightarrow aC$
C, a	$C \rightarrow aa$

The difficulty is with the pair of productions $A \rightarrow B$ and $A \rightarrow C$, both of which correspond to the selector pair (A, a). The parser cannot deterministically map (A, a) to a single production, hence is not LL(1). Both B and C derive strings beginning with "a", hence the PDA cannot decide between the productions $A \rightarrow B$ and $A \rightarrow C$ when "a" is the next input symbol.

4.3.2. LL(1) Grammar Transformations

We now consider a transformation that is often effective on a grammar containing a left-recursion, which (1) will preserve the semantic ordering of binary operations, and (2) will usually succeed in generating an LL(1) grammar.

Given a simple left-recursion of the form

$$A \rightarrow Ax \mid Ay \mid \ldots \mid w \mid z \mid \ldots$$

we first *stratify* this production set by introducing two new nonterminals B and C, as follows:

$$\begin{aligned} A &\rightarrow AB \mid C \\ B &\rightarrow x \mid y \mid \ldots \\ C &\rightarrow w \mid z \mid \ldots \end{aligned}$$

Note that B collects the strings past the A in the left-recursive A productions, and that C collects all the other A production right-hand parts. Then the productions $A \rightarrow AB \mid C$ are rewritten:

$$\begin{aligned} A &\rightarrow CA' \\ A' &\rightarrow BA' \mid \epsilon \end{aligned}$$

where A′ is another new nonterminal.

Here is an example, which yields the production set G_1 displayed in the selector table in figure 4.3, as based on grammar G_0:

The basis grammar is:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow a \mid (E) \end{aligned}$$

which is an obvious extension of G_0.

We first stratify the E productions, introducing the new nonterminals E″ and T′:

Write $T' \rightarrow + T \mid - T$, so that the E productions become:

$$E \rightarrow E\ T' \mid T$$

Then these two productions are rewritten as:

$$\begin{aligned} E &\rightarrow T\ E'' \\ E'' &\rightarrow T'\ E'' \mid \epsilon \end{aligned}$$

In a similar way, the T productions are rewritten:

$$F' \rightarrow * \ F \mid / \ F$$
$$T \rightarrow F \ T''$$
$$T'' \rightarrow F' \ T'' \mid \epsilon$$

The F productions remain unchanged. We clearly obtain the production set displayed in figure 4.3. The reader should verify that the resulting grammar is LL(1) by verifying the selection table given in the figure. Figure 4.4 displays a derivation tree for the sentence "a−a−a", which shows that the two operators are correctly associated.

Exercises

1. Design a table-driven LL(1) implementation, using the sparse table approach described in chapter 3. Write a table interpreter in some language of your choice.

2. Add unary minus, logical operators and indexed variables to grammar G_1, and attempt to transform it into an LL(1) grammar.

3. Construct a selector table for the grammar

$$S \rightarrow 0S0 \mid 1S1 \mid \epsilon$$

and trace the PDA for the strings 0110 and 0011.

4. Consider the LL(1) parser and grammar given in figure 4.3. Let the parser emit "LOAD a" on production $F \rightarrow a$, ADD on production $T' \rightarrow +T$, SUB on production $T' \rightarrow -T$, MPY on production $F' \rightarrow *F$ and DIV on production $F' \rightarrow /F$. Show through selected examples that the system emits valid reverse Polish code for arithmetic instructions, correctly associated and distributed.

4.4. Recursive Descent Parsers

The recursive descent parsers are closely related to the LL(1) parsers. They are among the most popular of the compiler parsers, perhaps because the parsing and semantics operations appear together as a reasonably lucid and self-explanatory program. The method uses procedure calls and other programming techniques familiar to most programmers. It is a top-down method, and we shall show that a necessary condition for a recursive descent

compiler to operate correctly is that its source grammar be LL(1). We shall explore two recursive descent synthesis methods. The first is an algorithm that accepts an LL(1) grammar and generates a recursive descent compiler program, in the form of a sequence of statements in an Algol-like language. The program produced by this method can be proven correct without much effort and illustrates the connection of recursive descent to LL(1) grammars; however, it is rather inefficient, as it exploits only a few of the control structures available in a high-level language.

The second recursive descent method accepts an extended BNF grammar. Such a grammar may contain closure, alternation, and concatenation operators, similar to those in regular expressions. With these features, a more satisfying recursive descent parser may be constructed. It, too, is related to the LL(1) grammars; however, the LL(1) concepts must also be extended to the new grammar form, and we shall show how this is done, and how the parser may be certified valid relative to its basis grammar.

The basic notion of recursive descent is that each nonterminal in a grammar is associated with a recursive procedure. The task of the procedure is to accept any string derivable from that nonterminal, and only such strings. It must furthermore move the read head through the string only if it accepts it, and not move the read head if it fails to accept it. Within such a procedure, the alternative right parts of the productions associated with the procedure are tested one at a time. It is essential that for any string in which one alternative succeeds, none of the others may succeed.

It can be seen from this brief introduction that a recursive descent parser is a top-down system, and that the central problem of validation for the parser is that the alternatives within a production set with a common left member be distinguishable through their first symbol. We shall explore this notion in greater detail upon formally defining a recursive descent parser generator.

Example of a Recursive Descent Parser. Figure 4.5 displays a recursive descent parser for an arithmetic grammar containing the four arithmetic operations. It consists of three procedures, EXPR, TERM, and FACT, and a Boolean variable FLAG. These procedures make use of a variable CHARACTER, which contains the next character in the input list, and a procedure NEXTCHAR, which fetches the next character. The input identifiers are assumed to be letters.

The procedures EXPR, TERM, and FACT must be recursive; that is, their return addresses must be stacked. Since they have no local variables or call parameters, nothing else need be stacked.

The task of procedure EXPR is to examine the input string beginning with the current symbol CHARACTER, and attempt to scan an arithmetic expression, where an *expression* consists of a term or a list of terms separated by the operators + and −. The task of procedure TERM is to identify a term, where a *term* is a factor or a list of factors separated by the operators *

```
 1  procedure EXPR; {recognizes expressions in add/subtract operators }
 2  begin
 3    TERM;
 4    if not FLAG then return;
 5    while FLAG do
 6    begin
 7      if CHARACTER = '+' then
 8      begin
 9        NEXTCHAR;
10        TERM
11      end
12     else
13      if CHARACTER = '-' then
14      begin
15        NEXT CHAR;
16        TERM
17      end
18     else FLAG: = FALSE
19    end;
20    FLAG: = TRUE;
21    return
22  end; (procedure EXPR)

24  procedure TERM; {recognizes expressions in * and /}
25  begin
26    FACT;
27    if not FLAG then return;
28    while FLAG do
29    begin
30      if CHARACTER = '*' then
31      begin
32        NEXTCHAR;
33        FACT
34      end
35     else
36      if CHARACTER = '/' then
37      begin
38        NEXTCHAR;
39        FACT
40      end
41     else FLAG: = FALSE
```

Figure 4.5. A recursive descent compiler for a simple arithmetic grammar, essentially G_0 extended with all four arithmetic operations.

```
42    end;
43    FLAG: = TRUE;
44    return
45  end; {procedure TERM}

47  procedure FACT; { recognizes operands and parenthesized
    expressions }
48  begin
49    if CHARACTER > = 'A' and CHARACTER < = 'Z' then
50    begin
51      NEXTCHAR;
52      FLAG: = TRUE
53    end
54   else
55    if CHARACTER = '(' then
56    begin
57      NEXTCHAR:
58      EXPR;
59      if not FLAG then return;
60      if CHARACTER = ')' then
61      begin
62        NEXTCHAR;
63        FLAG: = TRUE
64      end
65     else FLAG: = FALSE;
66    end
67   else FLAG: = FALSE;
68    return
69  end {procedure FACT}
```

Figure 4.5. (cont'd.)

and /. The task of FACT is to identify a factor, where a *factor* is a parenthesized expression or an identifier. Each of these moves the read head as far as possible, considering the form it is asked to identify, possibly making no move at all. The variable FLAG is set to TRUE if a match was possible, such that the read head moved, and set to FALSE if a match was not possible. The state of the FLAG upon emerging from a call on any of these procedures then indicates whether another alternative should be tested, or a syntax error has occurred, or whether the recognition was successful to this point.

An input string is parsed by setting CHARACTER to its first character, then calling EXPR. Procedure EXPR calls the other procedures and these may end up calling EXPR again recursively. Eventually, EXPR terminates

with FLAG set to TRUE if the string is an expression and FALSE otherwise. A termination with FLAG = FALSE is such that CHARACTER is the first character in error in the input string.

It is not obvious that these procedures operate correctly. We shall discuss this point later. For now, consider the string A*(B − C). A trace of the calls and operations within the procedures for this string is given in figure 4.6.

In the trace, EXPR first calls TERM, which first calls FACT. FACT then examines whether the first character is "(" or LETTER. An expression may only begin with one of them. If one or the other is seen, the next character is fetched by a CALL NEXTCHAR, which also deposits it in CHARACTER, and FLAG is set TRUE. Program control then returns to line 27 in procedure TERM.

TERM next sees if a "*" or a "/" is in CHARACTER. These symbols are legal after LETTER, but are not the only legal operators; so are "+" and "−". It happens that "*" is seen next. Thus TERM scans "*", advances past it (line 34), and calls FACT again.

The read head is now positioned at CHARACTER = "(". On "(", FACT calls EXPR and then looks for a ")". A failure to find EXPR, then ")" is a syntax error.

EXPR is therefore launched on the string "B − C)". It will scan the "B − C" portion, but will return on the ")". Note that it reports TRUE to the calling program FACT, line 58, so that the ")" will next be scanned.

Eventually, the input string is exhausted. It first happens in FACT, when the ")" is scanned. Technically, CHARACTER should contain $\perp$ when the end of the input string is reached, so that the remaining tasks will yield valid results. The result is an "unwinding" of the recursive calls, with the end result a return from EXPR with FLAG = TRUE.

Exercises

1. Trace the following strings through the program of figure 4.5, and indicate whether they are valid or invalid.

   (A)
   A+B*C
   A++B
   −C

2. Why is FLAG set to TRUE near the end of the EXPR routine?

3. What is the purpose of the WHILE loop in EXPR and TERM?

Remaining String	Procedure	FLAG	Line	Comment
A*(B-C)	EXPR	- - -	2	Enter procedure EXPR
	EXPR	- - -	3	Call TERM
	. TERM	- - -	26	Call FACT
	. . FACT	- - -	49	CHARACTER is LETTER
*(B-C)	. . FACT	TRUE	52	Input advanced, FLAG set TRUE
	. . FACT	TRUE	68	Return from FACT to TERM
	. TERM	TRUE	27	Return point in TERM
	. TERM	TRUE	28	Start WHILE-DO loop
	. TERM	TRUE	30	CHARACTER is "*"
(B-C)	. TERM	TRUE	33	Call FACT
	. . FACT	TRUE	49	CHARACTER is not LETTER
	. . FACT	TRUE	55	CHARACTER is "("
B-C)	. . FACT	TRUE	58	Call EXPR
	. . . EXPR	TRUE	3	Into EXPR, call TERM
	. . . . TERM	TRUE	26	Into TERM, call FACT
-C)	. . . . . FACT	TRUE	52	CHARACTER is LETTER
	. . . . TERM	TRUE	28	Start WHILE-DO
	. . . . TERM	FALSE	41	CHARACTER is neither "*" nor "/"
	. . . . TERM	TRUE	43	But that's OK
	. . . EXPR	TRUE	5	Start WHILE-DO
	. . . EXPR	TRUE	7	CHARACTER is not "+"
	. . . EXPR	TRUE	13	CHARACTER is "-"
C)	. . . EXPR	TRUE	16	Call TERM
	. . . . TERM	TRUE	26	Call FACT
	. . . . . FACT	TRUE	49	CHARACTER is LETTER
)	. . . . TERM	TRUE	28	Back to TERM, start WHILE-DO
	. . . . TERM	FALSE	41	Tests for "*", "/" failed
	. . . . TERM	TRUE	43	But that's OK
	. . . EXPR	TRUE	5	Another WHILE-DO try
	. . . EXPR	FALSE	18	Failed this time
	. . . EXPR	TRUE	20	But that's OK
	. . FACT	TRUE	59	Where we left off in FACT
	. . FACT	TRUE	60	Character is ")"
ϵ	. . FACT	TRUE	68	Returns TRUE
	. TERM	TRUE	28	Try another WHILE-DO
	. TERM	FALSE	41	Failed
	. TERM	TRUE	43	But that's OK
	EXPR	TRUE	5	Try another WHILE-DO
	EXPR	FALSE	18	Didn't work
	EXPR	TRUE	20	But that's OK

Figure 4.6. Trace of program of figure 4.5 for the string "A*(B−C)".

4. Suppose code is emitted upon scanning the operators "+", "*", etc., as in the following table. What kind of target machine would support the emitted code? Give some examples of expressions and the emitted code.

operator	emitted instruction
+	ADD
−	SUB
*	MPY
/	DIV
letter	LOAD letter

5. Extend the program of figure 4.5 to include replacement statements of the form

<variable> = <expression>

4.4.1. Construction and Validation

We shall now show that a recursive descent parser can be constructed automatically from any context-free grammar. Unfortunately, it will not necessarily recognize the grammar's language, and may possibly recognize no language at all by getting into an endless loop. We must therefore develop a condition that a grammar yield a valid parser, and this requirement will turn out to be exactly the LL(1) condition for a grammar.

Construction

Let $G = \{N, \Sigma, P, S\}$ be a context-free grammar. We construct a recursive-descent parser M from G as follows:

1. For each nonterminal symbol A in N, construct a recursive type Boolean procedure A, with no formal parameters or local variables.

2. Let FLAG be a global logical variable, whose value is TRUE or FALSE.

3. Consider procedure A, for some nonterminal A, and the set of A productions:

$$A \rightarrow w_1$$
$$A \rightarrow w_2$$
$$\vdots$$
$$A \rightarrow w_n$$

Each of the strings w_1, w_2, etc. will become separate programs with one entry point and one exit point. Then procedure A is written as follows:

```
procedure A: boolean;
begin
   (program for w1);
   if FLAG then return(TRUE)
  else
   (program for w2);
   if FLAG then return(TRUE)
  else
   (program for w3);
        .
        .
        .
   (program for wn);
   return(FLAG)
end
```

(*Note*: We return the value of a typed procedure through the RETURN statement. This is not a Pascal convention, but one that we find useful for discussion purposes here).

4. The program for a right member w of some production depends on whether w is empty or not, and if not, the sequence of terminal and nonterminal symbols of which it is comprised.

(a) If w is empty, and the procedure name is A, then the program for w is written:

```
FLAG:=MEMBER(CHARACTER, FOLLOW(A))
```

where CHARACTER is the character currently under the read head, FOLLOW(A) is the FOLLOW set for A, and MEMBER is a Boolean procedure that returns TRUE if CHARACTER is in FOLLOW(A) and FALSE otherwise.

(b) Let $w = a_1a_2a_3 \ldots a_r$, where the a_i are terminal or nonterminal symbols and $r \geq 1$. Then the program for w is written:

```
if MATCH(a1) then
begin
   FLAG:=TRUE;
   if not MATCH(a2) then ERROR;
   if not MATCH(a3) then ERROR;
```

```
        .
        .
        .
    if not MATCH(a_r) then ERROR
end
else FLAG:=FALSE
```

where MATCH(X) depends on whether X is a terminal or a nonterminal symbol. If X is nonterminal, then MATCH(X) is simply X, i.e. a call of procedure X. If X is terminal, then MATCH(X) is a function call COMPARE(X), where function COMPARE is:

```
procedure COMPARE(X: char): boolean;
begin
    if X=CHARACTER then
    begin
        NEXTCHAR;
        return(TRUE)
    end
    else return(FALSE)
end
```

Procedure NEXTCHAR fetches the next character, placing it in variable CHARACTER. Some special symbol must be used to indicate the end of the string, disjoint from the alphabet set.

ERROR is an error routine; when called, the machine will block and a syntax error is reported at the position of the read head in the input string.

5. Given some input string z. The parser is invoked by setting the read head to the left-most symbol of z, placing it in CHARACTER, then calling procedure S, which corresponds to the start symbol in G. Then if the grammar G is LL(1), procedure S returns TRUE if z is in L(G) and FALSE otherwise.

Example. Consider the productions

$$E'' \rightarrow T'\ E''$$
$$E'' \rightarrow \epsilon$$

which appear in the LL(1) grammar given in figure 4.3. They may be converted to the function E″ given below, by following the above rules:

```
procedure E'': boolean;
begin
   if T' then
   begin
      FLAG:=TRUE;
      if not E'' then ERROR
   end
   else FLAG:=FALSE;
   if FLAG then return(TRUE)
   else
   FLAG:=MEMBER(CHARACTER, FOLLOW(E''));
   return(FLAG)
end
```

Now consider the productions

$$F \rightarrow a$$
$$F \rightarrow (E)$$

which also appear in figure 4.3. These may be converted to the procedure F given below:

```
procedure F: boolean;
begin
   if COMPARE("a") then
   begin
      FLAG:=TRUE
   end
   else FLAG:=FALSE;
   if FLAG then return(TRUE)
   else
   if COMPARE("(") then
   begin
      FLAG:=TRUE;
      if not E then ERROR;
      if not COMPARE(")") then ERROR
   end
   else FLAG:=FALSE;
   return(FLAG)
end
```

These procedures are more complicated than they need to be. They may easily be rewritten to improve their form and efficiency. It is also possible to

devise a more elaborate constructor algorithm that examines larger contexts in the extended grammar derivation tree, and to choose more appropriate control structures for the parser. Despite these objections, this construction is useful as a means of proving correctness and demonstrating the relation of recursive descent to LL(1) parsers. It will be less easy to do for the extended recursive descent system described later.

Exercises

1. Finish the parser based on figure 4.3, and implement it on some computer. Try a variety of valid and invalid strings.

2. Construct an equivalent LL(1) grammar for the following context-free grammar and construct a recursive descent parser from it (the goal symbol is <stmt>):

```
<stmt> ::= EXEC
         | BEGIN <decl-list> ; <stmt-list> END
<stmt-list> ::= <stmt-list> ; <stmt>
         | <stmt>
<decl-list> ::= <decl-list> ; DECL
         | DECL
```

3. Construct a parser for the grammar

$$S \rightarrow Abc \mid aAcb$$
$$A \rightarrow \epsilon \mid b \mid c$$

We have shown earlier that this grammar is not LL(1). Find a string for which the parser fails and determine just why and how the parser fails.

Grammar Validation

The following two examples show that the parser constructed from a grammar may fail. Neither grammar given below is LL(1), but the two grammars illustrate the two kinds of failure that a recursive descent parser will exhibit for a non-LL(1) grammar.

Failure Example 1. Consider the grammar

$$S \rightarrow S0 \mid 1$$

which yields strings in the regular expression $1\{0\}$. The parser constructed for this grammar consists of a single procedure, as follows:

```
procedure S: boolean;
begin
   if S then
   begin
      FLAG:=TRUE;
      if not COMPARE("0") then ERROR
   end
  else FLAG:=FALSE;
   if FLAG then return(TRUE)
  else
   if COMPARE("1") then
   begin
      FLAG:=TRUE
   end
  else FLAG:=FALSE;
  return(FLAG)
end
```

By tracing the procedure for a simple example, e.g., the string "10", we soon encounter a problem. Upon calling S, S immediately calls itself! There is no movement of the read head or any conditional testing. These calls will continue until the return address stack space is exhausted in the computer, causing a failure of the parser regardless of the input string.

It is clear from the grammar that the difficulty lies in the left-recursive production $S \rightarrow S0$. It is also clear that any left-recursion, whether simple or not, will cause a failure of the system on some input string.

This difficulty cannot be cured by reordering the productions, i.e., writing the grammar

$$S \rightarrow 1 \mid S0$$

to yield the following program, which scans the leading "1" nicely, but then fails to scan any subsequent "0"s. Furthermore, given any string whose first symbol is not "1", the program again falls into an endless sequence of recursive calls with no movement of the read head:

```
procedure S: boolean;
begin
   if COMPARE("1") then
   begin
      FLAG:=TRUE
```

```
    end
    else FLAG:=FALSE;
    if FLAG then return(TRUE)
    else
    if S then
    begin
       FLAG:=TRUE;
       if not COMPARE("0") then ERROR
    end
    else FLAG:=FALSE;
    return(FLAG)
end
```

Failure Example 2. The following grammar also yields a parser which fails to recognize the grammar's language:

$$S \rightarrow A \perp$$
$$A \rightarrow B \mid C$$
$$B \rightarrow a\ C$$
$$C \rightarrow a\ a$$

This language consists of the two strings "aa⊥" and "aaa⊥". Note that this grammar contains no left-recursive productions. The reader is invited to construct the parsers for this grammar G_2 and also for a grammar G_2', in which the second production is

$$A \rightarrow C \mid B$$

Now consider these parsers, P2 and P2′, respectively. Given the string "aa⊥" in P2, procedure S is called, which calls A. A calls B (since B is written first in the alternation, it will be called first). Procedure B accepts the first character "a", then calls C. However, C expects two more characters "aa", but rather sees only "a⊥". It therefore fails, setting the flag FALSE, which is reported back to B. Then B reports FALSE to A, which then calls C as the other alternative. Again, C reports failure, since it expects "aa" and sees "a⊥". The failure of A is reported to S, which reports failure for the string. We therefore have a failure to accept a string in the language defined by the grammar.

An examination of the cause of this failure reveals that B should back up the read head on a failure. It scanned "a" before calling C and finding that C was in trouble. Rather than blindly reporting failure, it would seem that B should attempt to restore the read head position and try another alternative before reporting failure. Unfortunately, a read head backup cannot be

confined to one or two operations in general, but must be part of a general backtracking system.

We therefore conclude that the parser fails for grammar G_2. Let us see if it also fails for grammar G_2', in which the A productions are

$$A \rightarrow C \mid B$$

This time, consider the input string "aaa⊥". Procedure S first calls A, which calls C first. C accepts the suffix "aa", and reports TRUE to A, which reports TRUE to S. However, S now expects "⊥", but sees "a" instead. It therefore fails.

Here, we see that the difficulty lies in procedure S; it gave up too soon. Rather than report an error, it should tell procedure A to try another alternative, if it had any left. However, this again requires a generalized backtracking system. Recall, from chapter 2, that a general backtracking system requires an exponential time to parse certain sentences, or to detect a syntax error, and is therefore impractical.

Recursive Descent Validation

In this section, we show that if the grammar is LL(1), then the recursive descent parser will recognize exactly the language of its basis grammar.

We assert that, if the grammar G is LL(1), then:

1. Every procedure A will halt in finite time on any input string.
2. If $S \Rightarrow^* xAy \Rightarrow xwy$, then procedure A will accept w within the input list wy. By *accept* we mean that A will return TRUE, having advanced the read head exactly through string w.
3. If procedure A accepts a string w, and $|w| > 0$, then $A \Rightarrow^* w$ in G.
4. If procedure A accepts the empty string with the input string y (without moving the read head, of course), then there exists an empty production $A \rightarrow \epsilon$ in G, and a derivation $S \Rightarrow^* xAz \Rightarrow xz$ such that FIRST(y) is in FIRST(z).

Corollary. If these four assertions are true, then it clearly follows that procedure S (where S is the grammar's start symbol) accepts exactly the language L(G) if G is LL(1). Exact acceptance requires that S reports TRUE and exactly scans every string in the language, and also that S reports FALSE or calls ERROR on every string not in the language.

We first show property 1, by showing that A returns in a finite number of operations.

Note first that no procedure contains an internal loop. There are simply alternative tests involving COMPARE operations, FOLLOW operations and calls on other procedures, and a finite number of these exist. We may therefore count a procedure call as an operation. Now a procedure A may call as many as $n-1$ other procedures without any movement of the read head, where n is the number of nonterminals in the grammar. If it calls n procedures, then one of these must be A, and this implies the existence of a derivation $A \rightarrow^* A\ldots$, and G cannot be LL(1). Each of these $n-1$ calls may call at most $n-2$ procedures, etc., yielding at most $(n-1)! = (n-1)\times(n-2)\times \ldots \times 2$ calls without moving the read head. Upon exhausting these calls, the read head must move, or A returns, or else ERROR is called.

We see that in at most $(n-1)!$ calls, procedure A must complete its operations. (In practice, A completes its task in a much smaller number of calls. It requires a time proportional to the length of the string it scans). QED

Now consider property 2. Let string w be such that $S \Rightarrow^* xAy \Rightarrow^* xwy$. We shall show that function A accepts w within xwy by induction on the number of derivation steps in $A \Rightarrow^* w$. Let the first step be

$$A \Rightarrow x_1x_1 \ldots x_n \Rightarrow^* w_1w_2 \ldots w_n$$

where $x_1 \Rightarrow^* w_1$, $x_2 \Rightarrow^* w_2$, etc., and $A \rightarrow x_1x_2 \ldots x_n$ is in P.
If $n=0$, then this is an empty production, $A \rightarrow \epsilon$. Since $w=\epsilon$ is within a sentential form xwy, CHARACTER is in FIRST(y) and is in the FOLLOW(A) set, since xAy is a sentential form. Therefore function A returns TRUE without moving the read head.

If x_1 is a terminal symbol, then $x_1 = w_1 =$ CHARACTER. Some COMPARE operation in A will succeed; furthermore, since G is LL(1), there can be exactly one such COMPARE function for this symbol x_1. Also if this COMPARE must compete with some function call B for acceptance of a string beginning with w_1, B must return false. If B returned TRUE, it would imply the existence of a derivation

$$B \Rightarrow^* w_1 \ldots$$

by the inductive hypothesis, and it further implies, by the way the function A is constructed, that $A \Rightarrow B\ldots \Rightarrow^* w_1\ldots$ and also $A \Rightarrow w_1\ldots$ directly, which is a violation of the LL(1) condition. We therefore conclude that there must exist exactly one COMPARE test for w_1, and that no competing procedure call can succeed. If a FOLLOW test exists, by the LL(1) condition, it cannot succeed either, else there would be a conflict between FOLLOW(A) and $A \rightarrow w_1\ldots$.

If x_1 is a nonterminal symbol, then there must be a CALL x_1 among the alternatives in the A procedure. Given that w is derivable from A, and w_1 derivable from x_1, by the inductive hypothesis, CALL x_1 must accept string w_1, reporting TRUE.

Once the first element x_1 is identified—it must be identified by one of the alternatives in procedure A—the remaining elements $x_2, x_3, \ldots, x_n$ must be scanned by COMPARE or CALL operations, else an ERROR results. A COMPARE will scan a character; the CALL x_i, by the inductive hypothesis, will scan string w_i and report TRUE. Therefore A accepts w in the context of the sentential form xwy. QED

Next suppose that $|w| > 0$, and that A accepts w through some alternative based on a production $A \rightarrow x_1 x_2 \ldots x_n$, where the x_i are terminal or nonterminal (property 3.) Now each of the x_i must accept a portion of w, by the inductive hypothesis, either through a COMPARE, a FOLLOW, or a procedure call operation, within A. Also the successful operations each move the read head through 0 or more positions in w, in the order $x_1, x_2, \ldots x_n$. Therefore by the inductive hypothesis $x_1 \Rightarrow^* w_1$, $x_2 \Rightarrow^* w_2, \ldots x_n \Rightarrow^* w_n$, where $w = w_1 w_2 \ldots w_n$. Therefore $A \Rightarrow^* w$.

Consider property 4. Assume that A accepts some string w where $|w| = 0$. Then w had to be accepted with no movement of the read head. This requires either an immediate acceptance through the FOLLOW test, or a sequence of calls culminating in a successful FOLLOW test. Suppose the former. Then CHARACTER is matched against some member of FOLLOW(A), which implies a derivation

$$S \Rightarrow^* xAy \Rightarrow xy$$

where CHARACTER is also in FIRST(y).

Now suppose that $w = \epsilon$ is accepted through some chain of calls. By induction on the number of procedure calls required to accept w, given that calls on procedures B, C, . . . , M are required in that order to accept w, then there must exist a production $A \rightarrow BC \ldots M$ and the derivation

$$A \Rightarrow BC \ldots M \Rightarrow^* \epsilon$$

QED

4.4.2. Extended Grammars

The construction process given in the previous section is not an effective substitute for an LL(1) table-driven parser. It requires a transformation of a production set in general in order to obtain an equivalent LL(1) production set, and once this is done, there is no advantage in writing a recursive descent program for the production set.

By using an extended grammar, a more aesthetically satisfying recursive descent parser may be automatically constructed. However, such a parser may also fail to recognize the language defined by the grammar and requires validation. The validation rules are more complicated to express, and require a set of tree structures. We shall develop the parser generation process and discuss validation without attempting proofs of the algorithms employed.

An extended grammar consists of a set of extended productions of the form

$$A \rightarrow w$$

where w is an *extended structure* (as next defined) and A is a nonterminal; for each nonterminal A, there is exactly one such production in the grammar.

An *extended structure* is a string consisting of terminals, nonterminals, and the meta-symbols |, (,), [,], {, and } as follows:

- ϵ (the empty string) is an extended structure.
- If $x \in \Sigma$ (i.e., a terminal symbol), then x is an extended structure.
- If $X \in N$ (i.e., a nonterminal symbol), then X is an extended structure.
- If S is an extended structure, then each of the following are extended structures:

  (S), meaning S itself.
  { S }, *closure*, meaning zero or more concatenations of S.
  [S], *option*, meaning zero or one occurrence of S.

- If S_1 and S_2 are extended structures, then each of the following are extended structures:

  $S_1 \mid S_2$, *alternation*, meaning a choice of S_1 or S_2.
  $S_1\ S_2$, *concatenation*, with the usual meaning.

It is clear from the above structure rules that the right member w of each extended production $A \rightarrow w$ is a regular expression, except that both terminal and nonterminal symbols are employed in the expression. We have also introduced a new unary operator, *option*.

Example. Grammar G_0 may be written as an extended grammar as follows. The productions $E \rightarrow T$, $E \rightarrow E + T$, and $E \rightarrow E - T$ define a structure for E that looks like this:

$$E \rightarrow T\ \{\ (\ +\ |\ -\ )\ T\ \}$$

That is, any expression consists of a *term* followed by any number of $+$*term* or $-$*term* elements, including none. Note that E no longer appears in the right-hand part of the extended production.

An equivalent way of expressing these productions is

$$E \rightarrow T\ \{\ (\ +\ T\ )\ |\ (\ -\ T\ )\ \}$$

The equivalence is apparent from the distributive law of concatenation over alternation.

Similarly, the productions T → F, T → T * F, T → T / F define a structure for T that looks like this:

$$T \rightarrow F \{ (* \mid /) F \}$$

Finally, the productions F → (E) and F → a define the structure:

$$F \rightarrow lp\ E\ rp \mid a$$

where *lp* and *rp* stand for the terminal symbols "(" and ")", respectively. Since "(" and ")" are metasymbols, we cannot permit these as terminal symbols.

We therefore have the extended grammar G_0':

$$\begin{aligned} E &\rightarrow T \{ (+ \mid -) T \} \\ T &\rightarrow F \{ (* \mid /) F \} \\ F &\rightarrow lp\ E\ rp \mid a \end{aligned}$$

4.4.3. Construction and Validation from an Extended Grammar

Given an extended grammar G = (N, Σ, P, S), we may construct a recursive descent parser from it by following the plan given in figure 4.7.

In using figure 4.7, an extended structure is best represented by a tree, containing *concatenation*, *alternation*, *closure*, and *option* internal nodes, along with terminal and nonterminal leaf nodes. Figure 4.7 then specifies the recursive descent program components developed from such a tree in a preorder scan. For example, the structure

$$T\{(+ \mid -)T\}$$

is decomposed as follows:

- the concatenation of T with "{(+ | −)T}".
- the closure "{(+ | −)T}" of "(+ | −)T".
- the concatenation of "(+ | −)" with T.
- the parenthesized structure "+ | −".
- the alternation of + with −.

The corresponding tree structure is given in figure 4.8.

The concatenation of T with "{(+ | −)T}" yields the program:

```
(program for T);
if FLAG then
begin
   (program for "{(+ | −)T}" )
end
```

For each production A → w

Construct a recursive procedure as follows:

```
procedure A;
begin
  (program based on structure w)
end
```

For a nonterminal structure X

Construct the procedure call:

```
X     (e.g. call procedure X)
```

For a parenthesized structure (w)

Construct the program sequence:

```
(program for w)
```

For an option [w]

Construct the program sequence:

```
(program based on w);
FLAG: = TRUE;
```

For a terminal symbol structure "a"

Construct the statement:

```
if CHARACTER = 'a' then
begin
  NEXTCHAR; (get the next character)
  FLAG: = TRUE
end else FLAG: = FALSE
```

For a concatenation structure w1 w2

Construct the program sequence:

```
(program for w1);
if FLAG then
begin
  (program for w2)
end
```

Figure 4.7. Construction rules for a recursive descent parser, based on an extended grammar representation of the language.

For a closure structure { w }

Construct the program sequence:

```
FLAG: = TRUE;
while FLAG do
begin
  (program based on w)
end;
FLAG: = TRUE;
```

For an alternation structure w1 | w2

Construct the program sequence:

```
(program based on w1);
if not FLAG then
begin
  (program based on w2)
end
```

Figure 4.7. (cont'd.)

It in turn expands into the program:

```
T;    {call procedure T}
if FLAG then
begin
   FLAG:=TRUE;
   while FLAG do
   begin
      (program for "(+ | −)T" )
   end;
   FLAG:=TRUE
end
```

The program for "(+ | −)T" is, at the outermost level, a concatenation, and, further down, an alternation of two terminal symbols:

```
(program for "+ | −" );
if FLAG then
begin
   T
end
```

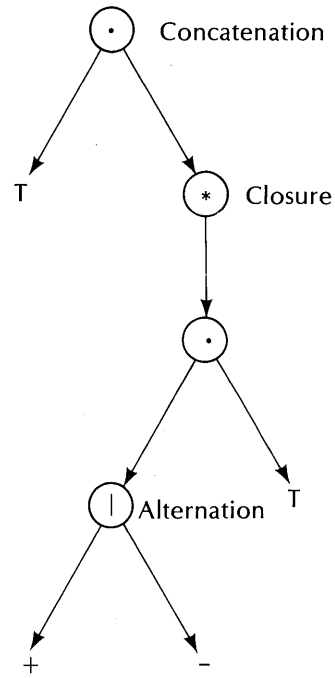


Figure 4.8. Tree for extended grammar structure T{(+ | −)T}.

Finally, the program for "+ | −" is:

```
if CHARACTER="+" then
begin
  NEXTCHAR;
  FLAG:=TRUE
end else FLAG:=FALSE;

if not FLAG then
begin
  if CHARACTER="−" then
  begin
      NEXTCHAR;
      FLAG:=TRUE
  end else FLAG:=FALSE
end
```

Putting all these together yields the program:

```
T;
if FLAG then
```

```
begin
   FLAG:=TRUE;
   while FLAG do
   begin
      if CHARACTER="+" then
      begin
         NEXTCHAR;
         FLAG:=TRUE
      end else FLAG:=FALSE;
      if not FLAG then
      begin
         if CHARACTER="-" then
         begin
            NEXTCHAR;
            FLAG:=TRUE
         end else FLAG:=FALSE
      end;
      if FLAG then
      begin
         T
      end
   end;
   FLAG:=TRUE
end
```

We need only embed the preceding statements in the procedure block:

```
procedure E: boolean;
begin
  (above program)
end
```

and we have a procedure that is equivalent to the EXPR procedure in figure 4.5. Its arrangement is slightly different, but the recognition logic is the same.

Finding FIRST and FOLLOW Sets for an Extended Grammar

A parser constructed from an extended grammar by the rules of figure 4.7 may or may not recognize the language of the grammar. In general, it will if the grammar is LL(1). However, we must formulate a more general LL(1) condition for an extended grammar. We can do so most conveniently if the grammar is represented as a set of trees, one tree for each nonterminal. The necessary tree structure is defined by the following rules:

1. For every nonterminal A in an extended grammar, construct a syntax tree. The root node will carry symbol A. The internal nodes other than the root will carry the grammar operations for alternation, concatenation, closure and option. The leaf nodes will carry a terminal or nonterminal symbol of the grammar or ϵ. Each tree represents a production in the extended grammar. A set of trees for grammar G_0' is given in figure 4.9. The symbols {., *, @, |} represent concatenation, closure, option, and alternation, respectively.

2. Each node will also carry two lists of terminal symbols, representing FIRST and FOLLOW associated with the node.

3. Construct a FIRST list for each node N as follows:

(a) If node N is associated with the nonterminal A, then locate the tree rooted in A; let its root node be R, and add the list FIRST(R) to FIRST(N), and add FIRST(N) to FIRST(R). This rule ensures that every node associated with a nonterminal symbol A, including a root node, carries the same list of FIRST symbols.

(b) If N is associated with the tree root R, then add FIRST(CHILD(R)) to FIRST(N).

(c) If N is associated with ϵ or a terminal symbol x, then FIRST(N) = $\{\epsilon\}$ or $\{x\}$, respectively.

(d) If N is associated with a closure, or an option, then add FIRST(CHILD(N)) to FIRST(N), and add ϵ to FIRST(N).

(e) If N is associated with a concatenation, then add

FIRST(FIRST(LEFTCHILD(N)) FIRST(RIGHTCHILD(N)))

to FIRST(N). LEFTCHILD(N) is the left child node, and RIGHTCHILD(N) is the right child node of N.

(f) If N is associated with an alternation, then add

FIRST(LEFTCHILD(N)) $\cup$ FIRST(RIGHTCHILD(N))

to FIRST(N).

4. Repeat step 3 until no more symbols can be added to any of the FIRST lists. A finite number of operations is required to reach that point. since the trees are finite and there are a finite number of symbols that may appear in any of the lists.

5. Construct the FOLLOW list for each node N as follows:

(a) If N is the root node, associated with a nonterminal A, then add FOLLOW(V) to FOLLOW(N) for each node V in each of the trees associated with the nonterminal A, and add FOLLOW(V) to FOLLOW(N). This rule ensures that every node associated with some nonterminal A carries the same FOLLOW list.

(b) If N is associated with the grammar's goal symbol S, add ϵ to

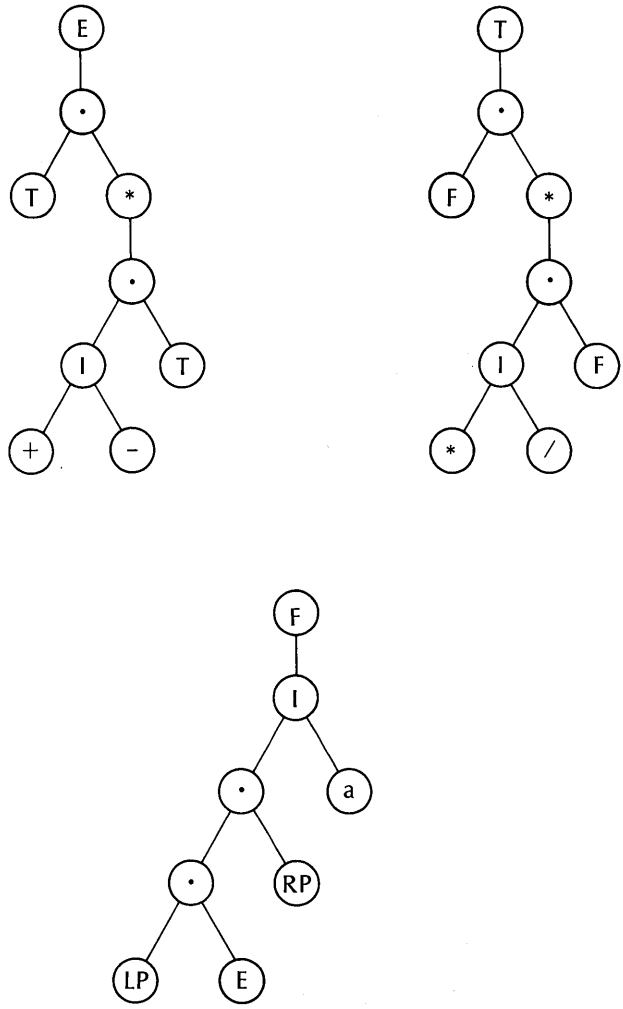


Figure 4.9. Syntax trees for grammar G_0'.

FOLLOW(N), and to the FOLLOW lists of all other nodes associated with S.

In the remaining rules, let F = PARENT(N), where N is not the root node:

(c) If F is a concatenation node, and N = LEFTCHILD(F), then add

FIRST(FIRST(RIGHTCHILD(F)) FOLLOW(F))

to FOLLOW(N).

(d) In all other cases, add FOLLOW(F) to FOLLOW(N).

6. Rule 5 is repeated until no more additions to the FOLLOW lists can be made.

The meaning of FIRST(X Y), where X and Y are lists, is

$$\begin{aligned} \text{FIRST(X Y)} &= (\text{X} - \{\epsilon\}) \cup \text{Y} && \text{if } \epsilon \text{ is in X, or} \\ &= \text{X} && \text{if } \epsilon \text{ is not in X.} \end{aligned}$$

Discussion

We shall not attempt a complete justification of these rules. That would require more development of extended derivations and extended sentential forms, etc. However, the notion of FIRST and FOLLOW is essentially that developed for simple grammars, except that both are lists of terminal symbols. A FIRST list is the set of all symbols that appear first in a sentence derivable from that node.

For example, if node N is an alternation, then its FIRST list should include FIRST(L) and FIRST(R), where L and R are its children, since a sentence derivable from N includes sentences derivable from L and from R. Similarly, if N is a concatenation, with children L and R, then the sentences of N are derivable from sentences of the form XY, where X is derivable from L and Y from R. Clearly, FIRST(N) includes FIRST(XY) = FIRST(L FIRST(R)).

If node N is a closure or option, then its FIRST includes FIRST(C), where C is its child, but also includes ϵ, since these structures may derive the empty string.

If node N is a terminal symbol, then clearly FIRST(N) contains only that symbol.

Finally, if node N is a nonterminal, it may be a root node, in which case FIRST(N) includes FIRST(C), where C is its child. We also ensure that all the nodes associated with a given nonterminal carry the same FIRST lists.

The FOLLOW rules essentially state that ϵ follows a goal symbol, and that FOLLOW lists propagate downward through the tree, except to left children of concatenation nodes. The FOLLOW set of a left child N of a concatenation is the FIRST list of N's right sibling, and if this list contains ϵ, also includes the FOLLOW of N's right sibling. Any FOLLOW members found for some nonterminal node V are copied to all the other nodes associated with V.

For example, consider the set of trees representing the extended grammar G_0':

$$\begin{aligned} &\text{E} \rightarrow \text{T } \{ \ (+ \mid -) \ \text{T}\} \\ &\text{T} \rightarrow \text{F } \{ \ (* \mid /) \ \text{F}\} \\ &\text{F} \rightarrow \textit{lp} \ \text{E} \ \textit{rp} \mid \text{a} \end{aligned}$$

The syntax trees for G_0' are shown in figure 4.9.

Figure 4.10 shows the same trees decorated with the FIRST lists. These lists were constructed from rule 3 above. A convenient place to start is some terminal node, for example *lp*, *rp*, or "a" in the F tree, using rule 3(c). The concatenation nodes in tree F clearly carry *lp* by rule 3(e) and the observation that the left children do not contain ϵ. The alternation node in tree F carries the union of the FIRST lists of its children, "a" and *lp*, by rule 3(f), and F carries its child's FIRST list by rule 3(b). The F list may now be entered in the other trees, in particular the T tree, which is similarly built up.

A partial FOLLOW list for the trees of figure 4.9 are shown in figure 4.11. For these, it is convenient to begin by adding ϵ to the goal symbol nodes E, then looking for concatenation nodes in which the right child is terminal or contains a nonempty FOLLOW list. Any additions to a FOLLOW list must then be propagated down the tree by rule 5(d). Thus the left T node in the E tree carries $\{+, -, \epsilon\}$ since these are in the FIRST set of its right sibling, a closure node. Similarly, $\{*, /, \epsilon\}$ are in the FOLLOW list of the left F node in the T tree. Since ϵ is in FOLLOW for E, it propagates down the E tree along its right. It cannot propagate to the left child of a concatenation node, however. Similarly, the set $\{+, -, \epsilon\}$ associated with any T node is propagated down the right side of the T tree, and eventually is added to the FOLLOW set of F. (We have not yet added to F the $\{*, /\}$ symbols found in the left node of the T tree.) Then the set $\{+, -, \epsilon\}$ found for F so far is also propagated down the F tree. However, these do not decorate the E node, since a concatenation node lies two levels above it.

Through repetitive applications of rule (5), the final FOLLOW lists appear as shown in figure 4.12. Note that the terminal nodes are also decorated. As a check on our work, these lists appear to be reasonable in terms of the known contexts of arithmetic operators. Any symbol in $\{*, /, +, -, \epsilon\}$ can follow an operand, as is clear from the following examples:

a*a a/a a+a a−a a

Similarly, either symbol in {a, (} can follow any of the arithmetic operators $\{+, -, *, /\}$ as is clear from these examples:

a*a a+(a−a)

Finally, a right parenthesis may be followed by any symbol in $\{*, /, +, -, \epsilon,)\}$ as is clear from the following examples:

(a)*a (a)/a (a)+a (a)−a (a+a) ((a+a))

The FOLLOW lists for the nonterminal symbols are less obvious; these lists and the lists for the internal nodes depend strongly on the grammar.

Parser Validation

A recursive descent parser constructed from an extended grammar is valid if the following two conditions on the syntax trees of the grammar are met:

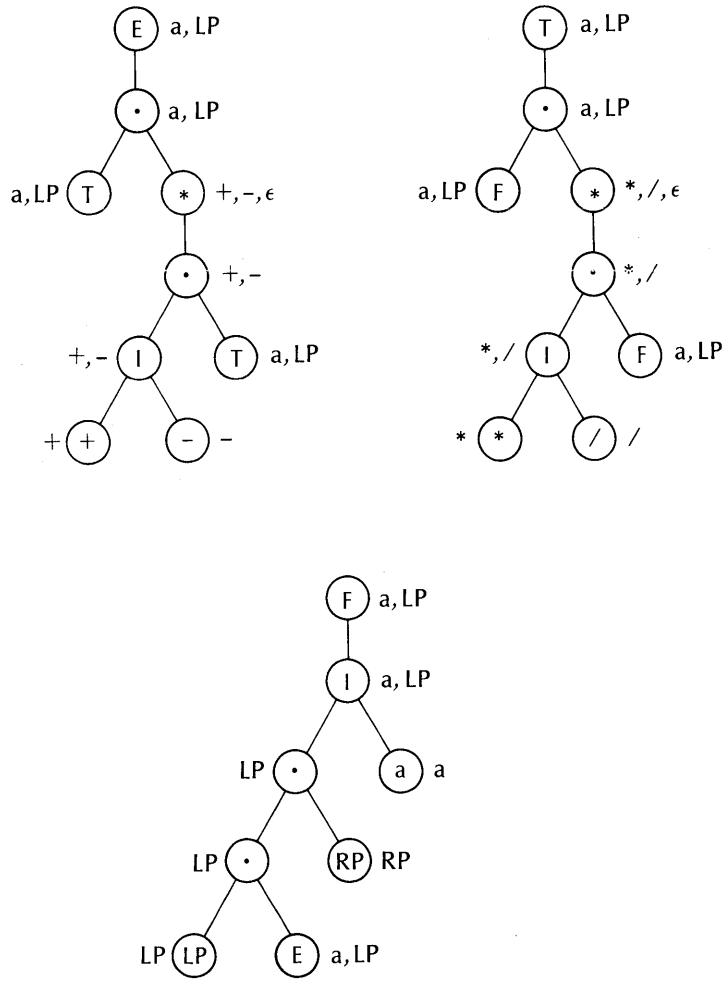


Figure 4.10. FIRST lists of the trees of figure 4.9.

1. For every closure or option node N,

$$\text{FIRST}(\text{FIRST}(\text{CHILD}(N))\ \text{FOLLOW}(N)) \cap \text{FOLLOW}(N) = \varnothing.$$

2. For every alternation node N,

$$\text{FIRST}(\text{FIRST}(\text{LEFTCHILD}(N))\ \text{FOLLOW}(N)) \cap$$

$$\text{FIRST}(\text{FIRST}(\text{RIGHTCHILD}(N))\ \text{FOLLOW}(N)) = \varnothing$$

We shall not attempt a formal proof of these assertions, but will discuss

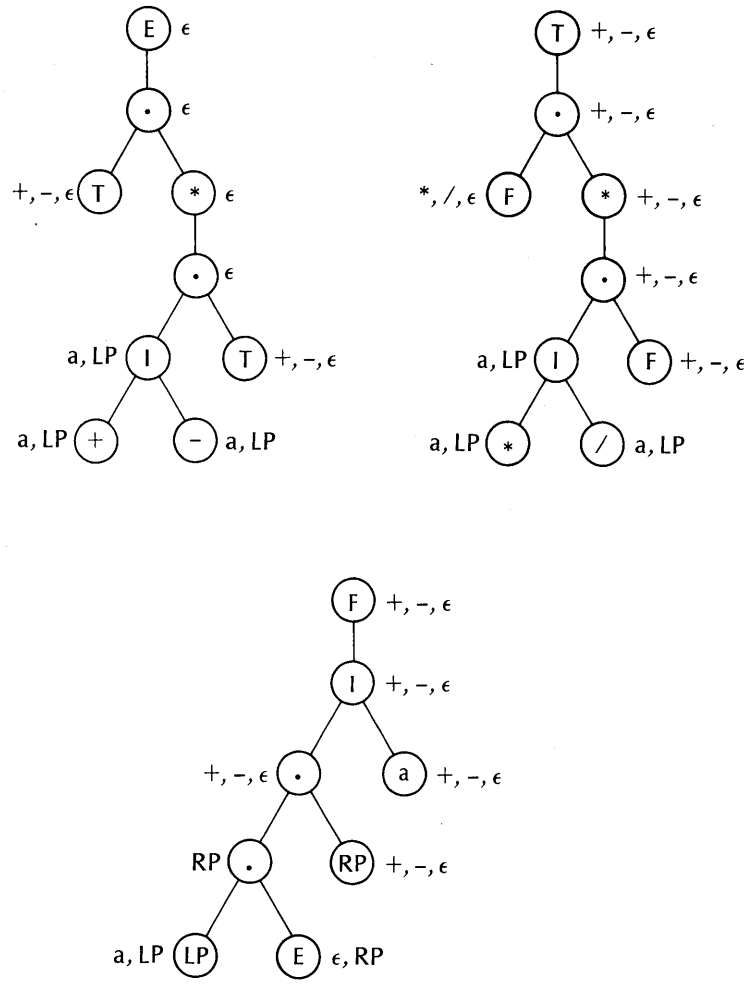


Figure 4.11. Partial FOLLOW lists for the trees of figure 4.9.

them informally. Note that these conditions are similar to the LL(1) conditions for a simple grammar. In an extended grammar, if the recursive descent parser fails to accept some input string, it will fail through either accepting some portion of the wrong alternative or by accepting a portion of a closure or option when the following string should instead have been accepted.

Rule 2 above guarantees that the parser can never choose the wrong alternative upon examining the next input symbol, since a pair of alternatives can only derive strings starting with disjoint symbols.

Rule 1 guarantees that when the parser attempts to recognize a closure or option, a following string cannot carry a conflicting FIRST symbol. That is,

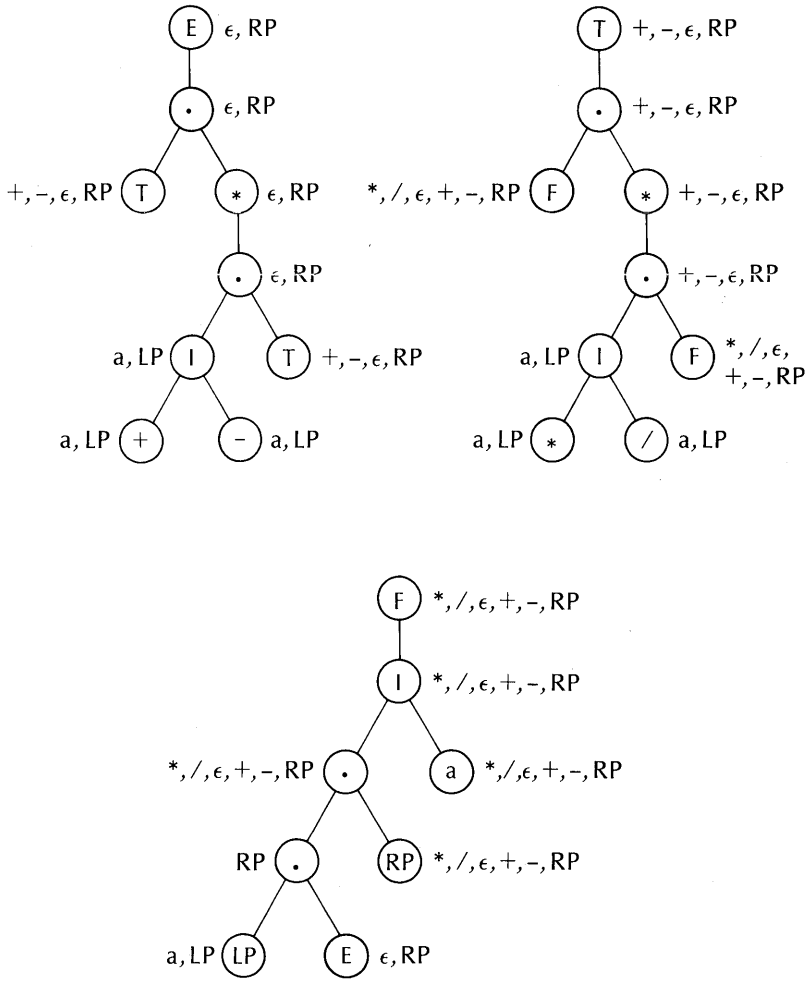


Figure 4.12. Complete FOLLOW lists for the trees of figure 4.9.

consider the form

$$\ldots\{x\}y$$

Rule 1 guarantees that the strings derivable from x cannot contain any leading symbol in common with the strings derivable from y. As a special case, x must not be able to derive an empty string ϵ, since then the parser cannot possibly determine whether a closure operation on ϵ is required or not. This case is covered by the rule, since if M (in the rule) can derive ϵ, then FIRST(M FOLLOW(N)) includes FOLLOW(N). Hence, if the parser attempts to accept a string derivable from x, and succeeds on the first symbol, then it cannot succeed on any first symbol in a string derivable from y. Also, if the

parser can scan a first symbol in a string derivable from y, then it must fail to accept any string in x, by failing on the first symbol in x.

Consider the decorated trees of Figs. 4.10 and 4.12. The alternation nodes in all three trees clearly satisfy the second condition, for example, in the E tree, the alternation node's left child carries $\{+\}$ and its right child carries $\{-\}$. Similarly, in the F tree, the children of the alternation node carry the disjoint sets $\{(\}$ and $\{a\}$.

The FIRST set of the child of the closure node in the E tree is $\{+, -\}$, while the FOLLOW set of the closure node is $\{\epsilon,)\}$. Similarly, the closure node in the T tree satisfies the validation condition. We conclude that a recursive descent parser constructed from these trees will correctly recognize the language of its extended grammar, which is essentially the parser given in figure 4.5.

Extended Grammar Represented as FSA

The structures in the right members of the productions of an extended grammar have the form of a regular expression, except that the machine transitions may be terminal or nonterminal transitions, and we will have a family of machines. Let us explore this matter further.

Construct a reduced DFSA for each of the right members of the extended productions in some grammar G. We will then have a machine M(A) for every nonterminal A in the grammar. Assume that the grammar is LL(1), i.e., it satisfies the validation conditions given in the previous section. Then we embed the machine in a recursive procedure as follows:

```
procedure A;
begin
   var STATE: integer;  {the current state}
   STATE:=0;      {the start state}
   {program to execute machine M(A)}
   FLAG:=if SUCCESS then TRUE else FALSE;
   return
end
```

The variable STATE is a local variable and is therefore stacked upon calling any other procedure, then unstacked upon returning from it. STATE indicates the state of machine M(A), which may be organized as a program or as an interpreted table. If organized as a table, then only one interpretative procedure for all the machines need be utilized, provided that it too, is recursive.

The machine transitions are of two kinds: a terminal transition, and a nonterminal transition. A terminal transition results in accepting the present symbol in the machine's present state, or reporting failure. If the terminal is

accepted, the read head is advanced one symbol; otherwise it is not moved.

On a nonterminal transition, the machine simply calls another machine by stacking its state and calling the appropriate procedure. The called machine either reports success or failure. If success, then this machine may continue, otherwise it must report failure.

A given machine continues until it must report failure or until it has reached a halt state and is unable to scan the next symbol; in this latter case it may report success.

The value of this approach lies in the attractiveness of an extended grammar in describing a language. For example, an extended grammar is closely related to a syntax graph, with which a number of computer languages are defined. The size of the parser system will likely be considerably smaller than if it were coded as a recursive descent system. However, it may be more difficult to attach semantic operations to the machine transitions, since these may no longer be clearly related to the extended grammar structure operations.

4.5. Bibliographical Notes

LL(k) grammars were first defined by Lewis and Stearns (Lewis [1968]). The theory of LL(k) grammars and their relation to deterministic parsers was extensively developed by Rosenkrantz and Stearns (Rosenkrantz [1970b]). Other papers on LL(k) grammar theory are Aho [1972a], vol. I, chapters 5 and 8; Kurki-Suonio [1969], and Griffiths [1974b]. Recursive descent parsers and compiler systems have been considered by a large number of authors. They form the basis of many so-called *compiler-writer* systems. A representative set of papers is Irons [1961], Metcalfe [1964], Feldman [1968], Gries [1971] (chapter 4), Aho [1972a] (vol. I, chapter 6), Griffiths [1974c], and Wirth [1976c].

CHAPTER 5

BOTTOM-UP PARSING AND PRECEDENCE PARSERS

A bottom-up parser reconstructs a right-most derivation in reverse, conceptually starting with a sentence in the language and ending with the goal symbol. We have previously examined bottom-up parsing as a tree-building process. The key to the process is the identification of the handle in any given right-most sentential form and a production that belongs with the handle in that form.

A bottom-up parser can always be constructed for a context-free grammar. It consists, as for the top-down parser, of an input list, a finite control, and a push-down stack. In general, it is nondeterministic, and there may or may not exist a lookahead system through which a deterministic parser may be constructed. There also exist grammars for which no deterministic parser can be constructed.

There are two broad classes of deterministic bottom-up parsers– the so-called *precedence* parsers and the *LR(k)* parsers. Of these two, the LR(k) parsers are considerably more powerful; they accept a much larger class of grammars. The LR(k) grammars also include all the LL(k) grammars, and constitute the largest class of deterministic grammars with left-to-right parsing and a k-symbol lookahead. Consequently, given any context-free grammar, it is more likely to have an LR(k) parser than any other.

A bottom-up parser appears to be more complicated than a top-down parser. The parsing theory is somewhat more difficult to follow than for a top-down parser, and the semantic operations seem to be less obvious. We can combat this feeling only by studying a complete compiler system. We shall present not only the bottom-up parsing theory and system, but a generalized semantics system as well, with enough rules so that anyone should be able to design a complete compiler from a set of objectives. Bottom-up semantics may be used with any bottom-up parsing method. The parser need only produce an ordered list of production numbers.

5.1. Nondeterministic Bottom-up Parsing

A bottom-up, nondeterministic parser differs from a top-down, nondeterministic parser in two respects:

1. The stack is more conveniently oriented with its top on its right end. The stack then holds some left suffix of a sentential form at any given

time. The handle will appear on the top of the stack just before an apply action.

2. The push-down automaton is extended by permitting a string of zero or more symbols to be popped from the stack in one operation. Recall that in a top-down PDA, it was never necessary to pop more than one symbol.

3. A NDPA constructed from a grammar will have two states. Recall that a top-down automaton contained only one state. However, the added state is used only to provide an elegant halt condition.

An *extended PDA* is a 7-tuple $P = (Q, \Sigma, H, \delta, q_0, \perp, F)$, where:

- Q is a set of *states*.
- Σ is a finite *input alphabet* (the tokens of the language).
- H is a finite *stack alphabet*.
- δ is a *finite mapping* from a 3-tuple in $Q \times H^* \times (\Sigma \cup \{\epsilon\})$ to the finite subsets of $Q \times H^*$.
- q_0 is a *start state* in Q.
- $\perp$ is an *initial stack symbol*, in H.
- F is a set of *halt states*.

Except for δ, this PDA is exactly like the top-down automaton introduced earlier. In this PDA, the mapping function δ considers the present state (in Q), but it may consider the symbol under the read head, and it may consider a string of 0 or more tokens on the top of the stack. In any case, δ is finite; it must be representable as a finite table. Through δ, the state, a top-of-stack string, and the next input token are mapped (in one move) to a new state and the stack top string is replaced by another string.

A *configuration* is written as follows. The stack top is at the right end of the stack:

(state, stack, input list)

Then a *move* is defined as a transformation of one configuration into another, as follows:

$$(p, zx, bw) \vdash (q, zy, w)$$

where $b \in \Sigma \cup \{\epsilon\}$, x, y, and z are in H^*, $w \in \Sigma^*$, and where

$$\delta(p, x, b) \text{ contains } (q, y)$$

The PDA is *nondeterministic* if some $\delta(p, x, b)$ contains more than one pair, or in other ways, e.g., through an empty x or b.

As before, the *language* defined by P, denoted L(P), is the set

$$\{w \mid (q_0, Z_0, w) \vdash^* (q, x, \epsilon) \text{ for some } q \text{ in } F \text{ and } x \text{ in } H^* \}$$

Note that the stack need not be empty to halt, and that the PDA is defined in such a way that it may make additional moves after its stack is empty.

Equivalence of Extended PDA's and Context-Free Languages

As before, we may define an extended PDA that recognizes the language of any context-free grammar. However, this one can reasonably be said to operate by handle recognition and replacement, making it a bottom-up parser.

Let (N, Σ, P, S) be a context-free grammar G. Then we construct an extended PDA R such that $L(R) = L(G)$ as follows:

1. R will have two states, $\{q, r\}$.
2. R's input alphabet is Σ.
3. The stack alphabet H consists of $N \cup \Sigma \cup \{\perp\}$.
4. The initial stack symbol is $\perp$.
5. The initial state is q, and the halt set $F = \{r\}$.
6. The mapping δ is defined as follows:
   (a) *Shift rule*: For every terminal symbol b in Σ, $\{(q, b)\}$ is a member of $\delta(q, \epsilon, b)$. These moves have the effect of shifting terminal symbols from the input source into the stack top. For example, the configuration (q, x, bw) would go to the configuration (q, xb, w) in such a move.
   (b) *Apply or reduce rule*: For every production $A \rightarrow w$ in P, $\delta(q, w, \epsilon)$ contains (q, A). These moves have the effect of taking a handle w on the stack top and reducing it to a nonterminal symbol A. Thus, a configuration (q, xw, u) would go to (q, xA, u) in such a move.
   (c) *Halt rule*: $\delta(q, \perp S, \epsilon)$ contains (r, ϵ), where S is the start symbol. This move can only occur once for a given parse and results in a halt. Note that it requires a stack containing only "$\perp S$", since there is no way for any stack symbols to appear beneath the $\perp$.

Such a parser is sometimes called a *shift-reduce* parser, since most of its moves are based on (6a) or (6b).

An Example. Consider the arithmetic grammar G_0, given below.

$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow a$

The PDA mapping for this grammar is then as follows:

$\delta(q, \epsilon, a) = \{(q, a)\}$ from rule 6(a)
$\delta(q, \epsilon, +) = \{(q, +)\}$
$\delta(q, \epsilon, *) = \{(q, *)\}$
$\delta(q, \epsilon, \text{“(”}) = \{(q, \text{“(”})\}$
$\delta(q, \epsilon, \text{“)”}) = \{(q, \text{“)”})\}$

$\delta(q, E+T, \epsilon) = \{(q, E)\}$ from rule 6(b)
$\delta(q, T, \epsilon) = \{(q, E)\}$
$\delta(q, T*F, \epsilon) = \{(q, T)\}$
$\delta(q, F, \epsilon) = \{(q, T)\}$
$\delta(q, (E), \epsilon) = \{(q, F)\}$
$\delta(q, a, \epsilon) = \{(q, F)\}$

$\delta(q, \perp E, \epsilon) = \{(r, \epsilon)\}$ from rule 6(c)

Although none of the mapping functions contains more than one pair, it is clear that the PDA is nondeterministic. The first set of five rules may be applied at any time, to shift another input symbol into the stack. Similarly, there may be opportunities to apply more than one of the second group of transitions. For example, with T*F on the stack top, the string T*F may be reduced to T, or F may be reduced to T.

Despite the great number of possible sequences of moves, the PDA recognizes exactly the language L(G). Most of the moves for a given string end in blind alleys.

Consider the string “a*a+a”. Only the following sequence of moves can reach the halt state:

1. $(q, \perp, a*a+a) \vdash$
2. $(q, \perp a, *a+a) \vdash$
3. $(q, \perp F, *a+a) \vdash$
4. $(q, \perp T, *a+a) \vdash$
5. $(q, \perp T*, a+a) \vdash$
6. $(q, \perp T*a, +a) \vdash$
7. $(q, \perp T*F, +a) \vdash$
8. $(q, \perp T, +a) \vdash$

9. $(q, \perp E, +a) \vdash$
10. $(q, \perp E+, a) \vdash$
11. $(q, \perp E+a, \epsilon) \vdash$
12. $(q, \perp E+F, \epsilon) \vdash$
13. $(q, \perp E+T, \epsilon) \vdash$
14. $(q, \perp E, \epsilon) \vdash$
15. (r, ϵ, ϵ)

Now consider the seventh configuration, $(q, \perp T*F, +a)$. It is possible to apply a shift rule to this, yielding $(q, \perp T*F+, a)$. However, no rule provides for reducing a stack with "+" on top, so we must again shift, yielding $(q, \perp T*F+a, \epsilon)$. At this point, we may reduce "a" to F, yielding $(q, \perp T*F*F, \epsilon)$, and then reduce F to T and T to E, yielding $(q, \perp T*F*E, \epsilon)$. However, we are at the end of the line for this particular sequence of moves. No further reductions or shifts are possible, and we have failed to reach the configuration $(q, \perp E, \epsilon)$ that permits a halt. There have also been no other possible moves. We choose this example to illustrate the central point—this PDA recognizes exactly the language defined by the grammar G. A proof follows.

Proof: We prove a somewhat strong result, which can easily be seen to imply that the PDA language and L(G) are equivalent:

Lemma 5.1. $S \Rightarrow^* xAy \Rightarrow^+ zy$ if and only if $(q, \perp, zy) \vdash^* (q, \perp xA, y)$, where the derivation is right-most. A is a nonterminal.

The "only if" part may be proven by induction on the number of steps n in the second sequence of derivations. The lemma states that, given a right-most sentential form xAy, in which A is the right-most nonterminal, and such that xA derives a terminal string z, then the PDA must be able to accept string z through shifts and reduces, and reach a stack that contains xA. The string y is unimportant, except within the inductive assertion of the proof.

For $n = 1$, string x must be empty, and a production $A \rightarrow z$ exists, where z is a terminal string. There clearly exist a sequence of shift moves for the terminal string z, yielding the moves:

$$(q, \perp, zy) \vdash^* (q, \perp z, y)$$

Note that z may also be empty. Then the stack top z may be reduced to A by the PDA construction rules, yielding the result

$$(q, \perp z, y) \vdash (q, \perp A, y)$$

which was to be proven, for the basis.

Now suppose the lemma true for any number of derivation steps less than n and that $xAy \Rightarrow^+ zy$ in n steps. The second sequence of derivations begins with

$$xAy \Rightarrow xwy \Rightarrow^* zy$$

where $A \rightarrow w$ is a production. Clearly, $xw \Rightarrow^* z$ in less than n derivation steps. Now xw may consist solely of terminals, which reduces this to the case $n = 1$. Hence suppose xw contains at least one nonterminal, say B; we also let B be the right-most nonterminal in xw, without loss of generality; then

$$xw = rBs$$

where s is a terminal string. We now have

$$S \Rightarrow^* xAy \Rightarrow xwy = rBsy \Rightarrow^* zy = usy,$$

since the terminal z must contain s as a suffix. But by the inductive hypothesis,

$$(q, \perp, zy) = (q, \perp, usy) \vdash^* (q, \perp rB, sy)$$

Then by some shifting rules,

$$(q, \perp rB, sy) \vdash^* (q, \perp rBs, y)$$

Next, since $rBs = xw$,

$$(q, \perp xw, y) \vdash (q, \perp xA, y)$$

QED.

A proof of the "if" part is left for an exercise.

Exercises

1. Construct a bottom-up NDPA for the following grammar G_0':

$$E \rightarrow E+E \mid E*E \mid a \mid (E)$$

   This grammar is ambiguous. Demonstrate that its NDPA will accept the string $a+a*a$ by either of the two derivations for it in G_0'.

2. Complete the proof of lemma 5.1.

Deterministic Bottom-up Parsing

The PDA constructed from a grammar G described in the last section is nondeterministic for several reasons:

1. A shift rule can always be applied as long as there exist remaining symbols in the input list.
2. Although a reduction rule can only be applied when the top of a stack string matches a right part of some production, there may be several

different productions whose right parts match a stack top string. Thus, for a stack top string xy, any production A → y might be applied through a reduction rule to yield xA. Note that the length of y is not defined by the PDA.

3. As a special case of point 2, an empty production A → ϵ could conceivably be applied at any time, since it requires no stack top match at all. The mapping associated with such a production is δ(q, ϵ, ϵ) contains (q, A), i.e., nonterminal A is pushed onto the stack.

There are two commonly used bottom-up, deterministic PDA—the precedence parsers and the LR(k) parsers. Each of them has a number of variations that may be employed to increase its power or to reduce its size.

5.2. Precedence Parsing

A precedence parser identifies the handle within a right-most sentential form without identifying the production. A *simple precedence parser* associates one of the three relations, "<", ">", or "=" with each pair of adjacent tokens in a sentential form. For a suitable grammar, the handle always becomes delimited between "<" and ">", and within it, "=" applies to the pairs. Finally, in the string preceding the handle, "<" or "≐" applies, but not ">". The handle may therefore be identified by scanning a sentential form from left to right until ">" is seen, then from right to left until "<" is seen.

A typical precedence table is given in figure 5.1, for a simple grammar G_2:

$$\begin{aligned} E &\rightarrow E - T \mid T \\ T &\rightarrow (E) \mid a \end{aligned}$$

Every sentence is preceded and followed by a special symbol "⊥", in order that the first and last pairs have a meaningful precedence relation. The table of figure 5.1 contains one entry for which two relations hold: {(, E} is in both of the relations "<" and "=". This is called a *conflict* and means that the parser is nondeterministic.

For example, in grammar G_2, within the sentential form ⊥((E−T))⊥, the adjacent pairs carry the following precedence symbols:

$$\perp < (< (\leq E = - = T >) \ldots$$

since E − T is the handle of this sentential form. The "≤" conflict for {(, E} indicates that "(E−T" could be a handle; however, this fits no production right part and therefore cannot be a handle.

The pairs in the string past the handle are not important, since the object is

Left \ Right	–	(	)	a	E	T	⊥
–		<		<		=	
(		<		<	≤	<	
)	>		>				>
a	>		>				>
E	=		=				>
T	>		>				>
⊥		<		<	<	<	

G_2: $E \rightarrow E - T \mid T$
$T \rightarrow (E) \mid a$

Figure 5.1. Simple precedence relation table for grammar G_2, shown.

to delimit the handle. Once the handle is reduced by applying a production (in this case, $E \rightarrow E+T$), a new sentential form is obtained, and its handle may similarly be found.

Although this appears to be an impossibly weak system, it is sufficiently powerful to parse (with a few special problems) most common programming languages, such as Algol, Fortran, and Basic. We shall therefore develop simple precedence in more detail.

A parsing table can be systematically constructed from a basis grammar. If the parsing table is free of conflicts, and no two productions have the same right member, then the grammar is a *simple precedence* grammar, and the precedence parser will work correctly. If conflicts exist, it may be possible to extend the precedence relations to include more than one symbol, or it may be possible to resolve the conflicts by modifying the parsing method slightly. Otherwise, the grammar is either ambiguous or unsuitable as a basis for a precedence parser.

Exercises

1. Find the handle in each of the following sentential forms, using the precedence table, figure 5.1:

⊥(E)a⊥
⊥((a−a))⊥
⊥T⊥
⊥(E−T)⊥

2. Show that each of the strings in exercise 1 are sentential forms by (a) displaying a derivation, and (b) by reducing the handle, finding the next handle, etc., until ⊥E⊥ is obtained. Resolve the "(, E" conflict in favor of the longest handle.

3. What happens upon parsing each of the following strings using figure 5.1? Are these sentential forms in G_2?

⊥(−a)⊥
⊥E)⊥

5.2.1. Relations

In order to discuss the precedence parsers in more detail, we need some background development.

A *relation* on two sets P and Q is some set of ordered pairs (x, y) such that x is in P and y is in Q. We may write

$$x \; R \; y$$

to indicate that the ordered pair (x, y) is in relation *R*. We may also write: $y \in R(x)$. An example of a relation is equality on pairs of integers; I = J is true if the integer I is the same as J and false otherwise. The members of the integer equality relation are (1, 1), (2, 2), ... Another example of a relation is the set of productions in a grammar G; P is a subset of ordered pairs drawn from the sets N (nonterminals) and $(N \cup \Sigma)^*$. This relation is indicated "→".

A relation *R* is said to be *transitive* if (A, B) in *R* and (B, C) in *R* imply that (A, C) is in *R*, which may also be written:

$$\text{if } (A \; R \; B \text{ and } B \; R \; C) \text{ then } A \; R \; C$$

A relation *R* is said to be *reflexive* if, for every A in *R*, (A, A) is in *R*. A relation *R* is said to be *symmetric* if (A, B) in *R* implies that (B, A) is in *R*.

A relation may be expressed as a matrix consisting of Boolean 0's and 1's. A "1" indicates that the corresponding row and column variables constitute a pair that is in the relation, and a "0" indicates a pair that is not in the relation.

The Boolean matrix for a symmetric relation is diagonally symmetric, i.e., an element at (i, j) in the matrix is equal to the element at (j, i). The matrix for a reflexive relation contains all 1's along its diagonal.

The *transitive completion* of a relation R, denoted R^+, is defined as:

1. R is in R^+.
2. If the pairs (A, B) and (B, C) are in R^+, then (A, C) is in R^+.

It can be seen that the inclusion of one pair through rule (2) may precipitate the inclusion of other pairs. The inclusion process must eventually terminate if R is a finite set.

The *reflexive transitive completion* of a relation R, denoted R^* is the union of R^+ and every pair (x, x) such that "x" is in R.

5.2.2. Boolean Matrix Sum and Product

We pointed out in the previous section that a relation can be expressed as a Boolean matrix. Sum and product operations on two matrices are convenient in constructing precedence tables.

The Boolean sum $P = R \vee S$ of two compatible matrices R and S is defined by

$$p(i, j) = r(i, j) \vee s(i, j) \qquad \text{for all } (i, j).$$

The Boolean product $P = R \wedge S$ of two square Boolean matrices R and S is defined as follows, where r(i, j) is a member of R, s(j, k) is a member of S, p(i, k) is in P, and the rank of the matrices is n:

$$p(i, k) = \bigvee_{j=1}^{n} (r(i, j) \wedge s(j, k))$$

The Boolean product is analogous to the algebraic product of matrices, except that a Boolean matrix contains only 0's and 1's. The product matrix element (i, k) is found by multiplying row i of matrix r by column k of matrix s; a bit-by-bit Boolean product is formed, and the resulting bit is the logical OR of these products.

An example Boolean matrix product is given in figure 5.2.

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

(a) **(b)** **(c)**

Figure 5.2. Boolean matrix product illustrated

Matrix Product and Transitive Completion of a Relation

Now the interesting result: the transitive completion of a relation R may be found by repeatedly replacing R by $R \vee (R \wedge R)$. We will develop a convenient and fast method of constructing precedence tables for a grammar on a digital computer.

This result is a consequence of the following theorem:

Theorem 5.1. Let X and Y be two square Boolean matrices representing the relations R and S, respectively. Then the pair (A, B) is in the product relation $X \wedge Y$ if and only if there exists an element C such that (A, C) is in R and (C, B) is in S.

Proof: The "if" part carries the postulate that C exists. Then matrix X has a "1" at the intersection of row "A" and column "C", and matrix Y has a "1" at the intersection of row "C" and column "B". Then the product matrix carries a "1" in row A, column B, and (A, B) is therefore in the product relation.

For the "only if" part, suppose the pair (A, B) is in the product matrix X Y. From the matrix product formula, there must be some row k in Y and column k in X such that there is a "1" at column i in Y and row j in X, otherwise the union could not be "1" for i = A and j = B. But this implies that (A, C) is in R and (C, D) is in S, for some C. QED

Warshall's Algorithm

Warshall [1962] found a fast means of computing the transitive completion B^+ of a relation B. His algorithm is expressed by the following Pascal program, where B is some given square Boolean matrix. Matrix B is tranformed into its transitive completion.

```
var I, J, K, N: integer;    {N is the rank of the matrix B}
type BOOLMAT = array [1..N, 1..N] of Boolean;
var B: BOOLMAT;

for I:=1 to N do
  for J:=1 to N do
    if B[J, I]=TRUE then
      for K:=1 to N do
        B[J, K]:=B[J, K] ∨ B[I, K];
```

Exercises

1. Consider the set of all living people in a monogamous society that forbids divorce. Classify each of the following relations on the pairs {A, B} as reflexive, symmetric, and/or transitive:

(a) A is an ancestor of B

(b) A is the mother of B

(c) A is a cousin of B

(d) (A is a father of B) or (A is a son of B)

(e) A lives in the same house as B

(f) A is married if and only if B is married

(g) A is single if B is single

(h) A is married if and only if B is single

2. Given a relation R expressed as a Boolean matrix, e.g,

```
var B: array [1..N, 1..N] of boolean
```

Write Pascal procedures that:

(a) Determine if R is reflexive.

(b) Determine if R is symmetric.

(c) Determine if R is transitive.

3. Consider a finite set of cities C with some interconnecting roads, and let relation *R* be defined by: "A *R* B if and only if there exists a road from A to B that passes through no other city." (Some of the roads are one-way). What is the significance of:

(a) *R* is reflexive.

(b) *R* is symmetric.

(c) *R* is transitive.

(d) The transitive completion of *R*.

Also explain how each of the following sets can be generated:

(e) *Trap* cities, that can be entered but not left,

(f) *Exit* cities, that can be left but not entered,

(g) The largest subsets C′ of C such that every city in C′ can be reached from every other city in C′.

5.2.3. Viable Prefix

A *viable prefix* is some prefix of a right-most sentential form that may include the handle, but no symbols past the handle. Thus if xhy is a right-most sentential form, such that h is the handle, then a viable prefix of xhy is any string r such that xh = rs, where s may be empty.

5.2.4. Precedence Pairs

An *(m, n) precedence pair*, where m and n are integers ≥ 0, is a pair of strings (x, y) such that $|x| = m$, $|y| = n$, xy is a substring of a right-most sentential form, and x lies wholly within a viable prefix. Note that string y may be within a viable prefix, or may lie partially or wholly within the following string. Since we demand that x have exactly the length m, and y the length n, we prefix every sentential form with m special symbols $\perp$, and suffix every sentential form with n special symbols.

For example, consider the sentential form ((E − T)) in grammar G_2, figure 5.1, and let (m, n) = (2, 1). Then we rewrite the form as $\perp\perp$((E − T))$\perp$ so that it is always possible to define a (2,1) precedence pair. Within this form, since E − T is the handle, the viable prefixes and (2, 1) precedence pairs are as follows.

Viable prefixes:

ϵ
(
((
((E
((E−
((E−T

(2, 1) precedence pairs:

$\{\perp\perp, ()\}, \{\perp(, ()\}, \{((, E)\}, \{(E, -)\}, \{E-, T)\}, \{-T,)\}$

5.2.5. Precedence Relations

An *(m, n) precedence relation* is a relation on the set of (m, n) precedence pairs defined on the right-most sentential forms within a grammar G. The three relations ($<$, $>$, $=$) are defined as follows:

1. A pair (x, y) is in the "$>$" relation if and only if $|x| = m$, $|y| = n$, and there exists some sentential form rhs such that $ux = rh$ and $yv = s$, and h is the handle, (r and s are assumed to include the necessary prefixes $\perp\perp\perp$.. and suffixes $\perp\perp\perp$...). That is, (x, y) marks the boundary position between the end of the handle h and the beginning of the following string s. However, x may include h, or h may include x. Note that y is a terminal string. The three dots ". . ." stand for "any string including empty".
2. A pair (x, y) is in the "$=$" relation if and only if $|x| = m$, $|y| = n$, and there exists some sentential form rhs such that $ux = rh'$ and $yv = h''s$, where the handle h is $h'h''$, and neither h' nor h'' is empty. That is, (x, y) marks a position within a handle, so that at least one tail symbol of x is in the handle and at least one head symbol of y is in the handle. Note that if the handle length is 1 or less, no "$=$" pair is defined for the handle.
3. A pair (x, y) is in the "$<$" relation if and only if $|x| = m$, $|y| = n$, and there exists some sentential form rhs such that h is the handle, $ux = r$ and $yv = hs$. That is, the pair (x, y) marks the beginning of the handle, such that the head of y is the head of the handle, and the tail of x is the tail of the string preceding the handle.
4. For every symbol X such that $S \Rightarrow *X\ldots$, the pair ($\perp \ldots \perp$, X) is in $<$. For every symbol X such that $S \Rightarrow * \ldots X$, the pair (X, $\perp \ldots \perp$) is in $>$. These special relation members are needed to cover the leading and trailing $\perp$ special symbols added to sentences. S is the start symbol.

5.2.6. Simple Precedence Grammar

If the grammar is such that no two productions have the same right member, then the grammar is said to be *uniquely invertible*. As we have seen, the precedence relations only identify the handle of a sentential form; they do not specify the production associated with that handle. Hence a precedence parser can be constructed for a grammar only if the grammar is uniquely invertible.

A grammar is said to be *(m, n) simple precedence* if and only if the relations "$<$", "$>$" and "$=$" are pairwise disjoint, and the grammar is uniquely invertible.

Example. Consider the simple grammar G_2:

$$E \rightarrow E - T \mid T$$
$$T \rightarrow (E) \mid a$$

Note that G_2 is uniquely invertible. The (1, 1) precedence relations "<", ">", and "=" are given in figure 5.1. We shall discuss systematic methods of generating such a table later. Let us first consider a short derivation to demonstrate that the precedence relations do in fact enable us to deterministically parse a sentence bottom-up. Consider the string (a − a) − a, which is in the language $L(G_2)$. The first group of relations, based on the table, are

$$\perp < (< a > - \ldots$$

(The table indicates that the pairs $\{\perp, (\}$ and $\{(, a\}$ are in the relation <, and that $\{a, -\}$ is in >). The handle is clearly "a", which can only be reduced to T. The reduction yields a new sentential form

$$\perp < (< T > - \ldots$$

which indicates that T is to be reduced to E, yielding

$$\perp < (< E = - < a >) \ldots$$

(The table contains a (<, =) conflict for the pair $\{(, E\}$. We shall deal with this matter next. For now, we select the correct choice by a nondeterministic oracle.) Note that although an "=" has appeared, the handle is "a", delimited by "< . . . >", which indicates a reduction of "a" to T, yielding

$$\perp < (< E = - = T >) \ldots$$

The handle here is E − T, which reduces to E. The next few steps are:

$$\perp < (= E =) > - \ldots \perp$$

$$\perp < T > - \ldots \perp$$

$$\perp < E = - < a > \perp$$

$$\perp < E = - = T > \perp$$

$$\perp < E > \perp$$

This terminal sentential form consists of the goal symbol bracketed by the special delimiter symbols $\perp \ldots \perp$.

Now let us examine the precedence table itself in the light of these definitions of a (1, 1) precedence relation. Each of the three relations may be determined (in theory at least) by examining all the possible right-most sentential forms. Unfortunately, there are usually an infinite number of them, so that this is hardly practical. Nevertheless, here are some derivations that bring out some of the table entries:

- $\perp E \perp \Rightarrow \perp T \perp$, therefore $\{\perp, T\}$ is in < and $\{T, \perp\}$ is in >.

- $\perp E \perp \Rightarrow \perp E - T \perp$, therefore $\{\perp, E\}$ is in $<$, $\{E, -\}$ is in $=$, and $\{-, T\}$ is in $=$.
- $\perp T \perp \Rightarrow \perp (E) \perp$, therefore $\{\perp, (\}$ is in $<$, $\{(, E\}$ and $\{E,)\}$ are in $=$, and $\{(, \perp\}$ is in $>$.
- $\perp T \perp \Rightarrow \perp a \perp$, therefore $\{\perp, a\}$ is in $<$, and $\{a, \perp\}$ in $>$.
- $\perp E - T \perp \Rightarrow \perp E - a \perp$, therefore $\{-, a\}$ is in $<$.

Exercises

1. Find derivations in G_2 that bring out the remaining precedence pairs of figure 5.1.

2. Consider grammar G_2. Show that the pairs ("(", $-$) and (a, E) cannot be in any of the three relations. Show that $(-, a)$ cannot be in "$=$" or "$>$". Hint: study the production tree for G_2.

3. Design efficient Pascal data structures and an algorithm for a precedence parser. The parser should return apply production numbers, given an input string, and halt on a syntax error.

Blank Entries

What about the blank entries in a precedence table? They apparently mean that the associated pair cannot appear and that therefore any symbol pair showing up in a parse that maps to a blank entry must be because of a syntax error. But how can we prove that a given entry is truly blank? We can be quite ingenious in finding sentential forms to fill in various table values, but we can never be quite sure that we have found them all. We therefore need a more systematic approach to precedence table generation than simply looking at various sentential forms. We take up this matter in section 5.2.7.

Three Theorems on Precedence Relations

Theorem 5.2. No grammar containing an empty production can be a precedence grammar.

Proof: Suppose a grammar G contains an empty rule, $A \rightarrow \epsilon$. Then some derivation

$$S \Rightarrow * \, xuAvy \Rightarrow xuvy \Rightarrow * \ldots$$

exists, where u and v are terminal or nonterminal symbols (not strings). Now the handle of the form xuvy is an empty string between u and v. By the

precedence relation definitions therefore, u $<$ v and u $>$ v, which creates a precedence conflict. QED

Theorem 5.3. Given any right-most sentential form xhy, where h is the handle, in a simple precedence grammar G, then either $<$ or $=$ holds between every pair within x.

Proof: Let xhy = ruvs, where u and v are some (m, n) precedence pair, and ru is contained in x. Thus (u, v) marks a boundary between symbols in x, or between x and the handle.

Consider the partial derivation tree T for the sentential form ruvs, and the nodes N_u and N_v. N_u is associated with the right-most symbol in u, and N_v with the left-most symbol in v. Then consider the upward paths in T from N_u and N_v; these must intersect in some node N_B associated with nonterminal B.

Suppose first that N_B is not the parent of N_u. There is then a subtree rooted in a child of N_B with a height of at least 1. It therefore contains the handle, contradicting the premise that the handle of ruvs is in vs. Therefore N_B is the parent of N_u.

If N_B is also the parent of N_v, we see that the pair (u, v) is in "=". If N_B is not a parent of N_v, it is nevertheless an ancestor. We then have a right-most derivation of the form

$$S \Rightarrow^* \ldots B \ldots \Rightarrow \ldots uV \ldots \Rightarrow^* \ldots uV' \ldots \Rightarrow \ldots uv \ldots$$

and by the precedence definitions, we see that (u, v) is in $<$.

Theorem 5.3 implies that a sentential form may be scanned from left to right through the string that precedes the handle. If the grammar is simple precedence, then only "$<$" or "$=$" applies between the pairs of this prefix string, and the scan may continue until the handle is found; it is delimited by "$>$" on its right end. Note that the fundamental precedence rule 3 does not specify precedence relations between the pairs in the string preceding the handle, and it might have turned out that some of these could be in "$>$" or in no precedence relation. Either case would be fatal to the prefix scanning operation. We now have assurance that the prefix scanning will work properly.

Theorem 5.4 Let $\perp^m w \perp^n$ be accepted by an (m, n) precedence parser. Then w is a right-most sentential form in G.

Proof: This theorem assures us that an (m, n) precedence parser accepts only strings in L(G). It should be obvious from the previous discussion that a string in L(G) is accepted by a (m, n) precedence parser, since the (m, n)

precedence relations are set up in such a way as to always recognize the handle.

We prove this by induction on the number of steps in the reduction process. For 0 steps, we must have $\perp^m S \perp^n$, where S is the start symbol, obviously a sentential form. Therefore assume the theorem true for all steps less than some k.

Let $\perp^m w \perp^n$ be accepted in k steps, and consider the first step in which the parser identifies the string h within w such that:

$$\perp^m x < y_1 = y_2 = \ldots = y_r > z \perp^n$$

where $h = y_1 y_2 \ldots y_1$. Here, x and/or z may be empty, and r may be equal to 1. In any case, the parser must see the relation ">" somewhere prior to or at the terminating $\perp^n$. Then, working backward through w from this point, marked by the end string z, it looks for the first "<", marked by the prefix x. In between, the relation "=" must apply. If "<" and then ">" are not found in that order, the string w is not accepted by the parser.

Now the string y so found must also be the right-hand part of some production $A \rightarrow y$ for acceptance; the nonterminal A is also unique because the grammar is uniquely invertible. The "=" relation applies among the members of some production right part, and the parser permits a reduction only if y matches a production right part. Hence the parser produces the new string $\perp^m xAz \perp^n$, which it accepts and which, by the inductive hypothesis, is a right-most sentential form or

$$S \Rightarrow * xAz$$

but $A \rightarrow y$, hence $S \Rightarrow {}^+ xyz = w$, which shows that w is a right-most sentential form. QED

Theorem 5.4 may also be stated: If a sentence w is not in L(G), then it is not accepted by the precedence parser. This error detection ability is obviously important, since we expect a parser to detect syntax errors as well as to analyze correct sentences. How are errors detected? The parser may encounter a precedence pair with an empty entry in its table, i.e., a pair (x, y) not in any of the three relations. It is also conceivable that it may isolate a string between "<" and ">" within some sentence that fails to match a production right member. We now give a simple example of a precedence grammar and a sentence which can only be rejected by a failure to match a delimited string to a production right part:

$S \rightarrow A \mid B$
$A \rightarrow a\ b\ c$
$B \rightarrow b\ c\ d$

It is easy to show that the (1, 1) precedence table is as follows:

	a	b	c	d	A	B	S	⊥
a		=						
b			=					
c				=				>
d								>
A								>
B								>
S								>
⊥	<	<			<	<	<	

so that there are no conflicts; the grammar is (1, 1) simple precedence. However, consider the string "abcd". The parser reaches this configuration:

$$\perp < a = b = c = d > \perp$$

which indicates that "abcd" is the handle, but it fails to match any of the production right parts. Furthermore, the string "abc" probably should have been reduced to A, yet the parser continued past it and failed to detect an error until the ⊥ was seen. (It is easy to construct grammars similar to this one in which the parser continues an arbitrary number of symbols past an error point).

Summary

Consider a precedence parser for a grammar G, with a table free of conflicts. Then:

1. Given a sentential form in the language L(G), the parser will accept the sentential form, reconstructing a right-most, bottom-up parse.

2. Given some sentence not in L(G), the parser must ultimately fail to accept the sentence; however, it may fail either:

   (a) by finding a symbol pair not in any of the relations $<$, $=$, or $>$, or

   (b) by failing to match the left-most string delimited by "$<\ldots>$" with the right part of any production.

It is the possibility of failing through case 2(b) that makes a precedence parser less desirable than an LR(k) parser, since it implies that a precedence parser may continue far past a reasonable error point before discovering an error. Nevertheless, much of this "weakness" depends on how one regards a sentence in error. A program with a syntax error should be rejected; no compiler that attempts to "read" the programmer's mind should be fully

trusted. The important question is not whether an error is detected early or late; it is whether the parser can somehow patch over the error and continue parsing in a reasonable manner, so that there is a fairly good chance of detecting other syntax errors. The patching over process is one form of *error recovery*. Error recovery issues are discussed in chapter 12.

5.2.7. Wirth-Weber Relations

The definitions of a precedence relation given in section 5.2.5 are not particularly useful as a means of obtaining the relations, since they are expressed in terms of the (usually) infinite set of sentential forms rather than the finite set of productions. We may restate the relations in an equivalent, but more useful form, called the *Wirth-Weber (1, 1) precedence relations*:

1. The relation $>$ is always defined on a pair (X, a), where "a" is a terminal symbol, since the symbol immediately to the right of any handle must be terminal. We say that $X > a$, or (X, a) is in $>$ if, for some production $A \rightarrow xBYy$ in P, $B \Rightarrow^+ \ldots X$ and $Y \Rightarrow^* a \ldots$.

2. A pair of symbols (X, Y) is in $=$ if there exists a production $A \rightarrow \ldots XY \ldots$ in P. Note that X and Y may be terminal or nonterminal symbols.

3. A pair of symbols (X, Y) is in $<$ if there exists a production $A \rightarrow xXBy$ in P such that $B \Rightarrow^+ Y \ldots$.

Equivalence Proof We now show that the Wirth-Weber relations are equivalent to the (1, 1) precedence relations given in section 5.2.5.

Consider the $=$ relation first. Since a handle is always the right member of some production rule, it follows immediately that the (1, 1) precedence relation "$=$" is equivalent to the Wirth-Weber "$=$" relation.

Next consider the $<$ relation, and let (X, Y) be in $<$ by the Wirth-Weber rules; then there exists a right-most derivation

$$S \Rightarrow^* uAv \Rightarrow uxXByv \Rightarrow^+ uxXYwyv$$

Here $B \Rightarrow^+ Yw$, and Yw is the handle of uxXYwyv. Then by the (1, 1) precedence relation (3), (X, Y) must be in $<$.

Now suppose that (X, Y) is in $<$ by the (1, 1) precedence relations. Then there exists a sentential form rhs such that h is the handle, X is the tail symbol of r, and Y the head symbol of h. Now this form has a derivation tree in which X and Y are leaf symbols (they may be nonterminal symbols). The upward paths from X and Y must intersect in some nonterminal symbol A, and A is associated with some production $A \rightarrow w$. Now X must be in w; that is, A is X's parent. Otherwise, X is derived from a nonterminal in w, and a handle would have to exist in r, contradicting the assumption that h is the handle in rhs.

Now Y cannot be in w, since the handle must be derived from Y. . . in the A production, not simply be a right part of it. These considerations, put together, imply the Wirth-Weber relation 3, namely, that X must be in a production $A \rightarrow xXBy$, and $B \Rightarrow^+ Y. . .$.

Finally, consider the $>$ relation. Given the Wirth-Weber relation 1, and production $A \rightarrow xBYy$ such that $B \Rightarrow^+ . . .X$ and $Y \Rightarrow^* a. . .$, then the string . . . X derived from B is a handle at some point in a derivation, and is followed by the terminal symbol "a". Therefore (X, a) is in the (1, 1) precedence relation. Then suppose that the sentential form rhs exists, where h is the handle, and let "a" be the first symbol in s and X the last symbol in h. Consider the derivation tree for this sentential form and paths up the tree from the leaf nodes X and "a". These paths must converge first on some nonterminal node A, associated with a production $A \rightarrow xBYy$, where the X path goes through B and the "a" path goes through (or is) Y. By arguments similar to those given for the $<$ relation, B must derive . . .X in at least one step, otherwise . . .X = h is not a handle. However, Y may be "a" or may derive "a. . .", which implies the Wirth-Weber relation ">". QED

Using the Wirth-Weber Relations

The Wirth-Weber relations are valuable in constructing a precedence table since they focus upon the productions and not upon sentential forms. However, we also need some relations that yield derived first and last symbols. We express them as relations in order to develop the simple precedence relations as matrix products.

LAST$^+$

$$(A \; LAST^+ \; X) \text{ if and only if } (A \Rightarrow^+ . . .X)$$

This relation defines the set of trailing symbols, terminal and nonterminal, derivable from A in one or more steps.

FIRST*

$$(A \; FIRST^* \; X) \text{ if and only if } (A \Rightarrow^* X. . .)$$

This relation defines the set of leading symbols, terminal or nonterminal, derivable from A in zero or more steps.

TFIRST*

$$(A \; TFIRST^* \; x) \text{ if and only if } (A \Rightarrow^* x. . . \text{ and } x \in \Sigma)$$

Symbol x is a member of the set of leading terminal symbols that can be derived from A, including A itself, if A is terminal.

TRANSPOSE

TRANSPOSE(M) is the transpose of matrix M, i.e., an element in row i and column j is moved to row j, column i, for all i and j.

Then the Wirth-Weber relations may be concisely defined in terms of the products of certain relations as follows ($\exists$ means *there exists*):

- The $=$ relation is found by inspection of the production set;

$$X = Y \text{ if and only if there exists a production } P \rightarrow uXYv$$

- The relation $<$ is equivalent to

$\{ (R, S) \mid (\exists A, B)(A \rightarrow ..RB.. \text{ and } B \Rightarrow^+ S...) \}$, which is equivalent to

$\{(R, S) \mid (\exists B)(R = B) \text{ and } (B \text{ FIRST}^+ S)\}$

Therefore $<$ is equivalent to $(=)(\text{FIRST}^+)$, viewed as a Boolean matrix product of $(=)$ and (FIRST^+).

- The relation $>$ is

$\{(R, a) \mid (\exists A, B, C)(A \rightarrow ...BC... , B \Rightarrow^+ ...R, C \Rightarrow^* a.., \text{ and } a \in \Sigma)\}$
Now $A \rightarrow ..BC..$ is equivalent to $B=C$, $B \Rightarrow^+ ..R$ is equivalent to B LAST$^+$ R, and $C \Rightarrow^*$ a. . . to C TFIRST* a. Therefore

$R > a$ is $(R \text{ (TRANSPOSE(LAST}^+))B) \wedge (B=C) \wedge (C \text{ FIRST* a})$

hence

$>$ is equivalent to (TRANSPOSE(LAST$^+$))$(=)$(TFIRST*),

viewed as a product of Boolean matrices.

The special symbols $\perp$ placed at the beginning and end of a sentence require special attention. The correct treatment is

1. Define a new nonterminal, G, and a new production

$$G \rightarrow \perp S \perp$$

where S is the grammar start symbol. Make G the new start symbol.

2. Include $(\perp, S)$ and $(S, \perp)$ in the $=$ relation.

3. Include $\perp$ in FIRST(G) and $\perp$ in LAST(G).

Now when the $<$ and $>$ relations are developed by matrix products, a conflict between $>$ and $=$ will be found in $(S, \perp)$. This conflict is spurious,

and one that the parser will never be concerned about. Delete the $=$ part of it. Also a conflict between $<$ and $=$ will be found in $(\perp, S)$; delete the $=$ part. Any other conflicts are real and must be dealt with. These two arise because we placed $(S, \perp)$ and $(\perp, S)$ in $=$, and we need to do so in order to develop $<$ and $>$ for the other symbols and $\perp$. However, S derives other strings, which means that $(\perp, S)$ will show up in $<$ and $(S, \perp)$ will show up in $>$. The parser must halt when the sentential form $\perp S \perp$ is found.

Example Figure 5.3 gives a worked-out example for a simple grammar, G_2, containing a binary "$-$" and parenthesizing. The basis grammar G_2 is given in figure 5.1.

Figure 5.3(a) shows the $=$ relation, easily found by inspection of the production set. The pairs $\{E, \perp\}$ and $\{\perp, E\}$ are in the relation because of the first production, $\{E, -\}$ and $\{-, T\}$ are in because of the second production, and $\{(, E\}$ and $\{E,)\}$ are in because of the fourth production.

Figure 5.3(b) gives the FIRST relation, easily found from the productions by inspection. Thus $\{G, \perp\}$ is in FIRST because of the first production; hence G FIRST $\perp$ is true, etc.

The transitive completion of FIRST is given in figure 5.2(c).

The reflexive transitive completion FIRST* of FIRST is given in figure 5.3(d). It is simply matrix (c) with the diagonal filled in with 1's.

The relation TFIRST*, which is all the members of FIRST* such that the second member of each pair is terminal, is given in (e).

Figure 5.3(f) gives the LAST relation. Thus $\{T,)\}$ is in LAST because of the production $T \rightarrow (E)$, etc.

The transitive completion of LAST is given in (g), and the transpose of $LAST^+$ is given in (h). This transpose is needed to form the $>$ relation, which is the product of three matrices, figure 5.3(k).

The $<$ relation is developed in figure 5.3(i) as the product of the $=$ relation and the $FIRST^+$ relation.

Finally, the complete precedence table, which is a summary of tables (a), (i), and (k), is given in (l). Except for the goal symbol row and column, it agrees with figure 5.1. We don't need to include a row and column for the goal symbol since the precedence parser halts on the stack contents $\perp G \perp$.

The $(<, =)$ conflict in $\{\perp, E\}$ and the $(>, =)$ conflict in $\{E, \perp\}$ should be replaced by $<$ and $>$ respectively, removing the spurious $=$ relation membership for these two pairs.

Direct Use of the Wirth-Weber Relations

The use of Boolean multiplication is well suited to a machine computation of an operator precedence table. It is not, however, at all suited to a pencil-and-paper computation. The Wirth-Weber relations may be used to determine an operator precedence table for a small grammar directly. We still

	−	(	)	a	E	T	G	⊥
−						1		
(					1			
)								
a								
E	1		1					1
T								
G								
⊥					1			

(a) =

	−	(	)	a	E	T	G	⊥
−								
(								
)								
a								
E					1	1		
T		1		1				
G								1
⊥							1	

(b) FIRST

	−	(	)	a	E	T	G	⊥
−								
(								
)								
a								
E		1		1	1	1		
T		1		1				
G								1
⊥							1	

(c) FIRST $^+$

	−	(	)	a	E	T	G	⊥
−	1							
(		1						
)			1					
a				1				
E		1		1	1	1		
T		1		1		1		
G							1	1
⊥							1	1

(d) FIRST*

	−	(	)	a	E	T	G	⊥
−	1							
(		1						
)			1					
a				1				
E		1		1				
T		1		1				
G								1
⊥								1

(e) TFIRST*

	−	(	)	a	E	T	G	⊥
−								
(								
)								
a								
E						1		
T			1	1				
G								1
⊥								

(f) LAST

Figure 5.3. Development of the precedence table for grammar G_1, using Boolean matrix operations.

	–	(	)	a	E	T	G	⊥
–								
(								
)								
a								
E			1	1		1		
T			1	1				
G								1
⊥								

(g) LAST $^+$

	–	(	)	a	E	T	G	⊥
–								
(								
)					1	1		
a					1	1		
E								
T					1			
G								
⊥							1	

(h) TRANSPOSE (LAST $^+$)

	–	(	)	a	E	T	G	⊥
–		1		1				
(		1		1	1	1		
)								
a								
E								1
T								
G								
⊥		1		1	1	1		

(i) $<\equiv$ (=) (FIRST $^+$)

	–	(	)	a	E	T	G	⊥
–								
(								
)	1		1					1
a	1		1					1
E								
T	1		1					1
G								
⊥								

(j) TRANSPOSE (LAST $^+$)) (=)))

	–	(	)	a	E	T	G	⊥
–								
(								
)	1		1					1
a	1		1					1
E								
T								
G	1		1					1
⊥								

(k) $>\equiv$ (TRANSPOSE (LAST $^+$)) (=) (TFIRST*)

	–	(	)	a	E	T	G	⊥
–		<	<	<		=		
(		<	<	<	⋖	<		
)	>		>					>
a	>		>					>
E	=		=					⋗
T	>		>					>
G								
⊥		<		<	⋖	<		

(l) Complete precedence table

Figure 5.3. (cont'd.)

need the functions TFIRST* , LAST+, and LAST* for each of the nonterminal symbols. Since there cannot be any empty productions, these are particularly easy to determine by inspection.

The = relation is easily found by inspection.

The < relation may be systematically found by finding all the occurrences of any symbol X followed by a nonterminal B in the right member of a production; let such a production be A→ xXBy. Then we include all the pairs (X, Y), where B FIRST+ Y, i.e., Y is in the FIRST+ set of nonterminal B.

The > relation may be systematically determined by finding all occurrences of a nonterminal B followed by any other symbol Y in a production. Then we take every symbol X such that B LAST+ X and every symbol x such that Y TFIRST* x and add (X, x) to >.

For example, consider grammar G_2, and the > relation. We find from the productions that we should consider the pairs:

B	Y	
E	⊥	from the first production G → ⊥E⊥
E	−	from the second production E → E−T
E	)	from the fourth production T → (E)

The relation E LAST+ X is valid for X = {T,), a}, and the symbols "x" are just {⊥, −,)}, hence the > relation must contain the pairs drawn from {T,), a} X {⊥, −,)}, and these are shown in figure 5.3(l).

Exercises

1. Use the Wirth-Weber relations directly to derive figure 5.1. from grammar G_2.

2. Construct a (1, 1) precedence table for grammar G_0, using the Wirth-Weber relations.

3. Write a Pascal program that generates a (1, 1) precedence table, given a grammar in suitable form. Begin by devising a suitable data structure to represent the grammar and other needed tables.

5.2.8. Other Precedence Parsers

A grammar is a *simple precedence* grammar if it is uniquely invertible and its simple precedence table contains no conflicts. As we have seen, even the simple grammar G_1 exhibits a conflict. As a grammar grows in complexity, more conflicts are likely to appear, and they may increase in severity.

The conflicts found in G_1 are between $<$ and $=$. When such conflicts only are found, they create a certain problem in locating the left-most end of a handle, but do not affect the identification of the right-most end. It is often possible to unambiguously fix the handle by some simple rule, such as "match the longest possible production right-hand member" when a conflict is seen. When it is possible to so identify a handle despite a ($<$, $=$) conflict, we say that the grammar is a *weak precedence* grammar.

A conflict with $>$ is much more serious; it means that the precedence table cannot deterministically locate the end of the handle. However, it may be possible to transform the grammar to remove the conflict or to consider a larger (m, n) in the precedence table.

A larger (m, n) than (1, 1) is impractical if the entire precedence table must be based on such strings. The size of an (m, n) table grows very rapidly with m and n, since it must consider a large number of possible combinations of m characters and n characters.

McKeeman, Horning, and Wortman [1970] proposed a secondary precedence table that applies only when the primary (1, 1) table contains a conflict in relation $>$. The secondary table is a (2, 1) table, which they find is sufficient to resolve most of the conflicts remaining in common programming grammars. Since the (2, 1) table need only consider the class of strings associated with the (1, 1) precedence pair found at the conflict, it need not be large. We shall not further consider the interesting question of deriving a secondary table, as the matter is thoroughly discussed in their text. Such a parser as this is called a *mixed strategy precedence* or *MSP parser*.

An *operator precedence* parser is constructed by considering precedence relations among terminal symbols only. The word "operator" arose through the consideration of grammars in which most of the terminals were in fact algebraic operators; operator precedence works rather nicely for them. Many compilers use operator precedence for arithmetic expressions and recursive descent for the other structures.

The construction of an operator precedence parser table is exactly like that for a simple precedence table, except that the nonterminals are ignored within the productions when looking for pairs of symbols. Grammar G_0 is a classic example of an operator precedence grammar. Its operator precedence relations are given in the following table:

	(	a	*	+	)	⊥
)			>	>	>	>
a			>	>	>	>
*	<	<	>	>	>	>
+	<	<	<	>	>	>
(	<	<	<	<	=	
⊥	<	<	<	<		

For example, "(" = ")" is obtained from the production F → (E), by ignoring E. The relation member "a" > "+" is obtained from the production E → E + T and LAST(E) contains "a". The precedence relations are generated essentially as before, except that the FIRST* and LAST* relations are redefined as follows:

- X FIRST* a if and only if X →* xay and x is empty or consists only of nonterminals.
- X LAST* a if and only if X →* xay and y is empty or consists only of nonterminals.

That is, we examine the sentential forms derivable from X, and ignore leading (in the case of FIRST*) or trailing (in the case of LAST*) nonterminals, picking out only the terminal symbols.

It is remarkable that algebraic expressions can be reduced to such a simple set of rules as conveyed by an operator precedence table. Such a table in fact conveys the popular notion of the *strength* or *hierarchy* of operators in a programming language. Precedence is a useful way of explaining to a neophyte programmer the ordering relations among various operators. For example, he may be told in a programming manual that "* has higher precedence than +". This means that a "*" production will be reduced prior to a "+" production, causing a multiply to be emitted first, whether "*" precedes or follows "+".

Now consider two operators with equal precedence, say + and −. The "equal" precedence means (in program terms) that the left-most of two operators are applied first, regardless of which kind it is. In parsing terms, the left-most production is reduced first.

In terms of productions, two operators, "@" and "#", have *equal precedence* if they are part of productions associated with the same left member:

E → E @ T
E → E # T
E → T

On the other hand, an operator "@" has *higher precedence* than "#" if it is derivable from a production containing "#", e.g.,

E → E # T
E → T
.
.
.
T →* .. @ ..

This production set is such that the T production containing "@" will be reduced prior to the $E \rightarrow E\#T$ production.

In this way, it is possible to design a grammar that contains any desired precedence or ordering relationship among its operators and other functions.

Exercises

1. Verify the operator precedence table given for grammar G_0.

2. Trace an operator precedence parse in G_0 for the following input strings:

   a + (a*a)
   a*a*((a) + a)

3. Examine the problem of identifying the handle and detecting syntax errors through failure to identify the handle, for an operator precedence parser.

4. Write an operator precedence parser in Pascal.

5. Sometimes a precedence table is such that we can assign a positive integer f(t) to each token t, where f(t) is such that

   f(a) < f(b) if and only if a < b
   f(a) = f(b) if and only if a = b
   f(a) > f(b) if and only if a > b

   How useful is f in a parser? Devise an algorithm that determines if function f exists, and apply it to the operator precedence table for G_0.

6. Design a grammar for a replacement statement language that contains these binary operators:

   $$+ \quad - \quad * \quad / \quad \uparrow \quad \text{AND} \quad \text{OR} \quad :=$$

   and parenthesizing. The precedences are

   highest AND
   OR
   ↑
   * / {* and / have equal precedence}
   + − {+ and − have equal precedence}
   lowest :=

   Also ↑ and := are to associate from right to left; all the other operators are to associate from left to right.

7. Consider the following productions in a grammar G:

$$S \rightarrow yAy \mid Cq$$
$$A \rightarrow Az \mid B$$
$$B \rightarrow tqz \mid B$$
$$C \rightarrow Ct \mid v$$

Is G a simple precedence grammar? If so, show that it satisfies the Wirth-Weber conditions. If not, how does it fail? Can you transform the grammar into an equivalent simple precedence grammar?

8. A string expression <sexp>, is defined by the following set of productions:

<sexp> ::= <sexp> : L / <cexp> | <cexp>
<cexp> ::= <cexp> * <prim> | <prim>
<prim> ::= <literal> | (<sexp>)
<literal> ::= L <literal> | ϵ

L is any letter. This grammar may be transformed into an operator grammar. How? What are the precedence relations among the terminal symbols of the grammar?

9. In many languages, the logical operators AND, OR and NOT and the arithmetic operators "+", "*", "/", "−", unary "−", are allowed to be mixed. For example, the assignment

$$a = a \text{ AND } b + c$$

is a valid assignment statement; it means the bit-by-bit AND of the variables a and b is to be interpreted as some number and added to c. Construct a grammar that includes these operators, such that an arithmetic operator has a higher precedence than any logical operation, AND and OR have equal precedence, and AND and OR defer to NOT. Develop the operator precedence table systematically for your grammar and demonstrate that these precedence relations in fact pertain.

5.3. Bibliographical Notes

The idea of using adjacent operators to control a recognizer was introduced intuitively by Perlis [1956]. It eventually became apparent that operator precedence was closely related to the language structure as expressed by its context-free grammar. Paul [1962] essentially solved a class of recognition problems in his thesis; his work did not become known in the U.S. for several years, however. Floyd's paper on operator precedence [1963] then became the first rigorous treatment of the problem of mechanically generating a parser given a context-free grammar. Wirth and Weber (Wirth [1966]) generalized Floyd's ideas into simple and weak precedence.

CHAPTER 6

BOTTOM-UP LR(K) PARSERS

6.1. LR(k) Grammars and Parsers

The LR(k) parsers comprise a family of bottom-up parsers that come as close to ideal left-to-right parsers as is theoretically possible. An LR(k) parser not only identifies a handle but also the production associated with the handle, with no additional decisions required of the rest of the compiler. Furthermore, it takes as much information as it possibly can from the portion of the program preceding and including the handle (the *viable prefix*). The only limitation is that it may examine at most k tokens in the input list past the handle.

An LR(k) parser for a grammar can always be constructed if any deterministic bottom-up parser for the grammar, limited to a k-symbol lookahead, can be constructed. This means that the largest class of grammars are covered by an LR(k) parser. Furthermore, an LR(k) parser covers every top-down LL(k) parser and can accept grammars not accepted by an LL(k) parser.

6.1.1. LR(k) Grammars

Let G be some grammar with start symbol S and consider a right-most derivation of a terminal string w in the grammar:

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w$$

Now consider a typical step in the derivation, as follows:

$$uAv \Rightarrow uxv$$

where $A \rightarrow x$ is the production used in this step, and uxv is one of the w_i or w itself.

We say that G is LR(k) if, for every such derivation and derivation step, the production $A \rightarrow x$ can be inferred by scanning ux and (at most) the first k symbols of v. Note that since A is the right-most nonterminal in uAv, v must be a terminal string.

Given that G is LR(k), we have several useful properties that make possible the development of a deterministic bottom-up parser:

1. The parser will know when to cease scanning a given sentential form uxv, i.e., it can detect the boundary between x and v.
2. The parser will be able to identify the handle x.
3. The parser will be able to uniquely select the production $A \rightarrow x$ that corresponds to the handle and to this sentential form. It happens that a grammar can be LR(k) and yet have several productions $A \rightarrow x$, $B \rightarrow x$, etc. with the same right member.
4. The parser will know when to halt.

As a totally impractical means of constructing a parser for an LR(k) grammar, we might somehow construct a large table that can map every string of the form uxv′, where v′ is a k-symbol head of v, to a production $A \rightarrow x$. Unfortunately, for most grammars, the table would have to be of infinite size, since the strings u can be indefinitely long. (However, the strings xv′ comprise a finite set.) We need a finite table for a practical compiler.

Knuth [1965] has shown that the set of all viable prefixes of right-most sentential forms can be recognized by a finite state automaton. This FSA can be used as the control machine in an LR(k) parser.

We first define an LR(1) parser. The "1" means that at most one token past the viable prefix will have to be examined in order to make any parsing decision. For most grammars, this token, called a *lookahead token*, is needed only for certain parsing decisions.

There are several different ways of constructing an LR(1) parser (or LR(k) in general), but every such parser operates the same way. The different construction means are classified as SLR(k), SLALR(k), LR(k), and LALR(k). The difference in the parser (if any) appears in the number of states and the size of a set of lookahead tokens when a k-symbol lookahead is required.

Each of the construction methods can be generalized to a k-symbol lookahead, although we shall develop only the case $k = 1$ in depth.

6.1.2. An LR(1) Parser

An LR(1) parser consists of an input list, a stack, and a finite control, essentially as required of any parser for a context-free grammar. The finite control in an LR(1) parser has a fairly large number of states. In a rough sense, the states keep track of vital information in the viable prefix.

Figure 6.1 shows an example finite-state control for grammar G_0. Notice that this machine contains transitions on both terminal symbols (a, +, *, (,), ⊥), and on nonterminal symbols (E, T, F, G). The parsing principle is quite simple—the finite control recognizes the viable prefix of any right-most sentential form, and falls into a state that uniquely identifies a production exactly when the handle is finished.

Three kinds of states exist in the machine of figure 6.1, *read* states, *lookahead* states, and *apply* states. In a read state, the parser must accept the

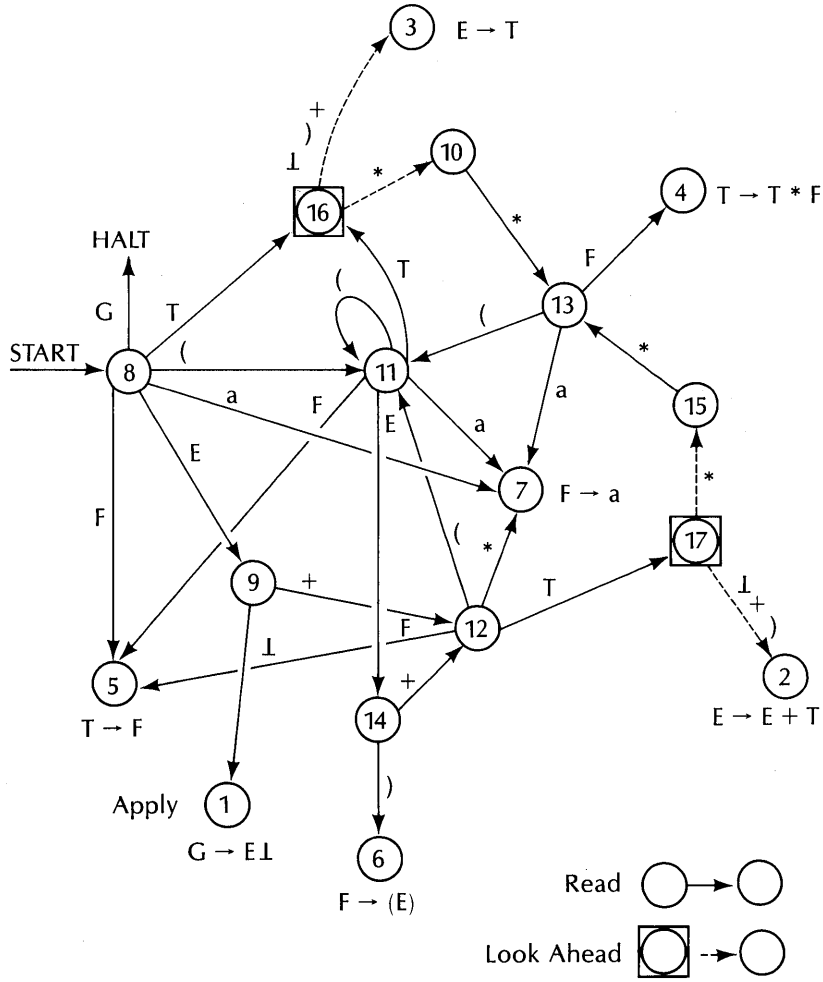


Figure 6.1. Complete LALR(1) parser for grammar G_0.

next token in the input list, shifting it to its stack, then move to some new state associated with the token. The read states in figure 6.1 are 8, 9, 10, 11, 12, 13, and 14. In a lookahead state, the machine "looks at" the next token, in order to make a state transition, but does not shift it into the stack. The lookahead states in figure 6.1 are 16 and 17. In an apply state, a production is indicated whose right member will be on top of the stack (it is the handle) and should be replaced by the left member of the production. At this point, the original sentential form has been scanned just past its handle, and the state is associated with a production.

The stack will contain the handle and the string preceding the handle upon falling into an apply state. Upon the reduction, the stack followed by the remaining input string comprises the next sentential form subject to a reduction. Therefore, conceptually, we construct a new input list consisting of the stack followed by the remainder of the old input list, and start over again. The apply states in figure 6.1 are numbered 1 through 7 and correspond to the production numbers.

In one cycle of the system, a right-most sentential form xwy is transformed into another one, xAy, such that

$$S \Rightarrow * xAy \Rightarrow xwy \text{ (right-most)}$$

Example. Let us trace the machine of figure 6.1 with a sentence, "a+(a*a)⊥". (A ⊥ is added to every sentential form to provide a uniform lookahead and halting symbol). The start state is number 8, and it indicates a transition on "a" to state 7, an apply state. The machine configuration at this point is like this:

stack: a

remainder of input list: +(a*a)⊥

State 7 indicates a reduction with the production F → a, hence the new machine configuration is

stack: F

remainder of input list: +(a*a)⊥

These two strings together comprise the next-to-last sentential form in a right-most derivation of "a+(a*a)⊥", i.e.,

$$G \Rightarrow * F+(a*a)\perp \Rightarrow a+(a*a)\perp$$

One cycle of the machine's operation is now complete. In the next cycle, we have the sentential form "F+(a*a)⊥", and we start over in state 8. This time, the indicated transition on F is to state 5, another apply state, which then yields the sentential form

$$T+(a*a)\perp$$

upon reduction. Then starting over with this in state 8, we are first led to the lookahead state 16 on symbol T. In this state, we should transfer to state 3, since "+" is a legal symbol along that transition path. The "+" is not shifted into the stack, however, which means that the apply state 3 operates on T and yields the next sentential form:

$$E+(a*a)\perp$$

When we start over again, the machine indicates a move on E to state 9, then on "+" to state 12, then on "(" to state 11, and finally on "a" to state 7. State 7 indicates that the handle is "a" and is to be reduced to F:

$$E+(F*a)\perp$$

is the resulting sentential form.

Figure 6.2 summarizes the complete parsing process on the string "a+(a*a)⊥". (We have numbered the variables "a" to distinguish them; the machine makes no such distinction.) It can be seen from the apply column in figure 6.2, which gives the indicated productions to be applied in each cycle, that the machine reconstructs a right-most parse of the sentence, in reverse order, but of course does so through a left-to-right scan.

The parsing process ends upon a move from the start state 8 to *halt*, which occurs on the goal symbol G. It may sometimes happen that a lookahead is required on a goal symbol move; but there must always be some halt state that indicates that the parse is successful.

The parser also rejects every invalid sentence, i.e., a sentence not in the language $L(G_0)$. Furthermore, it does so on the very first illegal symbol. Such an error detection must occur in a lookahead or a read state, never in an apply state, since these must accept or reject the next input symbol from the input list. When the parser is in some read or lookahead state, it can only accept a certain subset of the language's alphabet. For example, if in state 11, the parser may only accept one of the terminal tokens {a, (,)} next in the input list. (The nonterminal tokens don't count for error detection; why not?) Failure to see one of these must be a syntax error and may be so immediately reported.

The problem of dealing with an error will be discussed at greater depth in chapter 12. For now, we discuss a number of reductions on the parser operations and table sizes.

Parser Reduction

We first discuss a reduction in machine operations, making use of a push-down stack to achieve this. The stack will contain state numbers, which correspond to the symbols in a viable prefix of a sentential form.

Note first from figure 6.1 that every state has associated with it exactly one incoming symbol. For example, state 5 is associated with the *in-symbol* F, state 11 with "(", etc. This property is true for every LR(1) incremental parser, as we shall see upon defining the construction process. (It is not true in general for an arbitrary finite state machine.) We can therefore replace the symbols in the viable prefix of some sentential form by states with which they are associated (however, note that a given symbol may be the in-symbol of more than one state, e.g., E is an in-symbol for states 9 and 14). The list of states is effectively the states passed through during the scan of the sentential

Input string = $a_1 + (a_2 * a_3) \perp$

States	Apply	Stack	Input
8, 7	$F \rightarrow a_1$	F	$+ (a_2 * a_3)\perp$
8, 5	$T \rightarrow F$	T	$+ (a_2 * a_3)\perp$
8, 16, 3	$E \rightarrow T$	E	$+ (a_2 * a_3)\perp$
8, 9, 12, 11, 7	$F \rightarrow a_2$	$E + (F$	$* a_3)\perp$
8, 9, 12, 11, 5	$T \rightarrow F$	$E + (T$	$* a_3)\perp$
8, 9, 12, 11, 16, 10, 13, 7	$F \rightarrow a_3$	$E + (T * F$	$)\perp$
8, 9, 12, 11, 16, 10, 13, 4	$T \rightarrow T * F$	$E + (T$	$)\perp$
8, 9, 12, 11, 16, 3	$E \rightarrow T$	$E + (E$	$)\perp$
8, 9, 12, 11, 14, 6	$F \rightarrow (E)$	$E + F$	$\perp$
8, 9, 12, 5	$T \rightarrow F$	$E + T$	$\perp$
8, 9, 12, 17, 2	$E \rightarrow E + T$	E	$\perp$
8, 9, 1	$G \rightarrow E \perp$	G	
8, HALT		HALT	

Figure 6.2. Trace of LALR(1) parser of figure 6.1 for input a+(a*a)⊥.

form. For example, consider the sentential form "E+((a))", in which the handle is "a". Upon scanning "a", the stack will contain states 8, 9, 12, 11, and 7.

The right parentheses are not associated with states, since they are not yet part of a viable prefix. Upon reducing the handle, "a", to F, since this is indicated by state 7, we have the stack 8, 9, 12, 11, 5, which corresponds to the sentential form "E+((F))". We may view the correspondence of stack states and the viable prefix symbols as follows:

$$E_8 +_9 (_{12} (_{11} F_5))$$

Note that the state number (11) just preceding the handle (F) has not changed; it cannot, since it is associated with a symbol in the viable prefix preceding the handle. Clearly, we do not need to rescan the entire sentential form upon each reduction; it is only necessary to pick up at the state just preceding the handle. The states associated with the handle are popped, a new nonterminal is pushed, and a transition is made that depends on the state just below the handle.

The apply operation suggests another reduction. One nonterminal token is pushed in an apply, then the next state is determined by the (former) top-of-stack state and this nonterminal. We may therefore construct a table that maps a production number and the top-of-stack state (found after popping the number of states required by the apply) into a next state. With

such a table and operation, we no longer need any of the nonterminal moves in the read table.

Representation of the Parser as Tables

An efficient representation for an LR(1) parser consists of four tables and an interpreter. Three of these are shown in figure 6.3; the fourth table is a push-state table, and is unnecessary for the parser of figure 6.1.

The convention for this parser is that the stack contains every state except the "present" state.

The READ table yields a next state number, given a current state and a next symbol. The rule for entering the READ table is that the state must lie between 8 and 15, inclusive; then the next symbol is read and scanned; then the old state is pushed onto the stack. Note that we push the state we just left, not the one entered. A blank entry is a syntax error; errors are detected on the left-most symbol that cannot be a part of any sentential form.

The LOOKAHEAD table applies to the lookahead states 16 and 17; it is organized in a manner similar to the READ table, except that it is not necessary to list all the possible symbols involved in a given lookahead; hence the "else" column. We enter the table with a lookahead state and some next symbol in the input list. If the symbol is not found in the main body, then the "else" state is taken. The lookahead symbol is not scanned; eventually it will be scanned in a read state, at which time a possible syntax error will be detected.

The APPLY-GOTO table is actually two tables, entered by the state number 1 to 7, figure 6.3. The state number will always also be the production number. The second column in figure 6.3 (*pop*) is the number of symbols to pop from the stack before pushing another state number; this number is one less than the number of symbols in the right part of a production, except for an empty production, for which the pop number would be zero.

The GOTO portion of the APPLY-GOTO table contains the nonterminal transitions in the LR(1) parser machine of figure 6.1. Recall that upon identifying the handle (on top of the stack), the apply operation pops the handle and pushes a nonterminal symbol. Since we wish the stack to contain states, not symbols, and this is the only operation in which nonterminal symbols are pushed, we instead examine the state on the stack top just after the handle is popped. There must be a transition on this state and on the pushed nonterminal to some other state. The GOTO table contains the next state associated with the production state and the top-of-stack state. For example, suppose the machine falls into state 2. Then the POP column says that 2 states are to be popped. The top-of-stack state may then be 11 or some other state. If 11, then we go to state 14; if some other state, then we go to state 9. State 11 is left on the stack.

The GOTO table is obtained directly from the nonterminal transitions in

Read table for Go
1. Read
2. Push old state

	Symbol					
State	⊥	+	*	(	)	a
8				11		7
9	1	12				
10			13			
11				11		7
12				11		7
13				11		7
14		12			6	
15			13			

Lookahead Table
1. Look ahead

State	⊥	+	*	(	)	a	Else
16			10				3
17			15				2

Apply – Goto Table
1. Apply production
2. Pop stack
3. Go to

		Go to				
State	Pop	11	12	13	Else	Production
1	1				halt	G → E⊥
2	2	14			9	E → E + T
3	0	14			9	E → T
4	2		17		16	T → T * F
5	0		17		16	T → F
6	2			4	5	F → (E)
7	0			4	5	F → a

Figure 6.3. LALR(1) parser in table form.

figure 6.1 and reduced by lumping together the most popular top-of-stack state into an ELSE column, for each apply state. For example, states 2 and 3 both call for pushing the nonterminal E, therefore the E transitions in figure 6.1 belong in the GOTO table; these are

8 to 9,
11 to 14

The "11 to 14" transition is given directly, while the "8 to 9" transition is in the ELSE column. Similarly, for the apply states 6 and 7, the transitions on F are wanted; these are

8 to 5,
12 to 5,
11 to 5,
13 to 4

Since the most popular destination state is 5, the top-of-stack states 8, 11, and 12 are included in the ELSE column, while the "13 to 4" transition is specifically included.

In summary, the APPLY-GOTO table is entered with a state number that corresponds to a production. The right part of that production is on the stack top, and some semantic action may be taken at this point in the parse. After the semantic operations are complete, the stack is popped off the number of symbols in the POP column, and the stack top state P is used to enter the GOTO table for the current production, to yield a new state.

If the grammar contains any empty productions, a PUSH table is needed. Consider an APPLY state in the incremental parser for an empty production, and recall that the current state is not on the stack top (the previous state is). We therefore need another table called a PUSH table that maps an empty production apply state (the current state) to a next state. The stack is unaffected.

Example Parse with the Tables

Let us parse the string "a*(a+a)⊥" with the aid of figure 6.3. The start state is 8. Since 8 is a read state, we enter the READ table and scan symbol "a" in the input list; the table says the next state is 7. (See the first row in figure 6.4.) State 8 is pushed onto the stack.

State 7 is an APPLY state. Production 7 is F → a. Upon entering the APPLY table, it indicates a pop of zero states; the stack top is 8, hence the GOTO table indicates the next state is 5; the stack contains only 8 at this point.

State 5 is also an APPLY state, with a zero pop, production T → F, and the next state is 16.

State 16 is a lookahead state. The next symbol in the input list is "*", so the LOOKAHEAD table indicates the next state is 10. The stack still contains only 8, and only the first symbol, "a", has been scanned in the input list.

The remainder of the parsing operation is given in figure 6.4. The parsing ends upon entering state 1, which is associated with the goal production G→ E⊥. State 8 remains on the stack at this point.

Further Table Reduction

The tables in figure 6.3 may be reduced by using the sparse matrix techniques discussed in chapter 2, and exploiting identical rows in the original tables. For example, the READ table of figure 6.3 may be compressed to the following tables:

Trace of a * (a + a)⊥

State	Operation	Input	Pop	Push	Stack	Goto
8	R	a		8	8	7
7	A		0		8	5
5	A		0		8	16
16	L	*			8	10
10	R	*		10	8, 10	13
13	R	(		13	8, 10, 13	11
11	R	a		11	8, 10, 13, 11	7
7	A		0		"	5
5	A		0		"	16
16	L	+			"	3
3	A		0		"	14
14	R	+		14	8, 10, 13, 11, 14	12
12	R	a		12	8, 10, 13, 11, 14, 12	7
7	A		0		"	5
5	A		0		"	17
17	L	)			"	2
2	A		2		8, 10, 13, 11	14
14	R	)		14	8, 10, 13, 11, 14	6
6	A		2		8, 10, 13	4
4	A		2		8	16
16	L	⊥			8	3
3	A		0		8	9
9	R	⊥		9	8, 9	1
1	A		1		8	halt

Figure 6.4. Trace of LALR(1) parser tables of figure 6.3 for input a*(a+a)⊥.

S	N	Size
8	1	2
9	3	2
10	5	1
11	1	2
12	1	2
13	1	2
14	6	2
15	5	1

N	Symbol	Destination
1	(	11
2	a	7
3	⊥	1
4	+	12
5	*	13
6	+	12
7	)	6

For example, given read state 14, N is 6 and "size" is 2; the legal symbols and destination states are therefore in the second table starting at index N=6, running for 2 items; on "+" the next state is 12 and on ")" the next state is 6. Note that we have also compressed the similar rows 8, 11, 12, and 13 in the READ table into rows 1 and 2 of the reduced table. This kind of compression implies that the N values in the first table will not necessarily be monotonic increasing; we therefore also need the "size" column.

The lookahead and apply tables may similarly be compressed.

Exercises

1. Trace the parser of figure 6.1 on the strings

   (a + ((a * a)))⊥
   a * a * a⊥
   a * * a⊥
   a + a)⊥

2. Trace the parser of figure 6.3 on the strings of exercise 1.

3. Write an LR(1) parser in Pascal that interprets a set of tables similar to those in figure 6.3. Make it "semantic" driven, so that each call on a

parser procedure returns a production number; it in turn calls a scanner that returns the next token on each call.

6.1.3. LR(0) Parser Construction

We shall now describe a construction algorithm for an LR(0) machine. Such a machine will, in the absence of conflicts, parse sentences without ever requiring a lookahead. It will become the basis for the SLR(k) and SLALR(k) parsers.

The construction of an LR(k) machine will be defined in section 6.2. An alternative construction method for an LR(k) machine is described in section 6.3.

In this construction, each machine state is associated with a set of *items*, where an item is a production carrying a position marker. The rules for constructing the item-set corresponding to some state, and for starting new states, are reasonably simple and lend themselves to a machine implementation. The resulting parser machine is also in minimal form when the process is complete.

Recall that a viable prefix is a prefix of some right-most sentential form that includes no symbols past the handle of the form. We will place much emphasis on a viable prefix for one simple reason: the LR parser finite state machine recognizes viable prefixes, and at the same time a viable prefix is connected with a sentential form. It therefore serves as a kind of conceptual link between derivations (which involve sentential forms) and the LR parsing automaton.

Now consider an arbitrary state P in the complete LR machine, and the set (usually infinite in size) of viable prefixes associated with that state P. A viable prefix w is associated with state P if and only if the machine accepts w upon falling into state P. It should be clear that a given viable prefix can be associated with only one state, because we demand that the machine be deterministic. (We have not yet shown that it is finite.)

We say that two viable prefixes belong to the same LR equivalence class if they are associated with the same state.

The association of viable prefixes with states is an interesting thought, but not very useful, since each state may have an infinite number of viable prefixes associated with it. For example, all of the following viable prefixes in grammar G_0 are associated with state 12:

E+
(E+
((E+
T*(E+
T*((E+
(E+(E+

It is easy to find many more. We need only trace through the states starting at state 8 and ending at state 12. The loop on "(" in state 11 can be traversed any number of times, yielding another left parenthesis each time, as can the loop through 12-11-14. We obviously need to find some finite set that can be associated with each state.

Such a finite set is a set of items. An *item* is a marked production enclosed in brackets, i.e.,

$$[A \rightarrow x \,.\, y]$$

is an item if $A \rightarrow xy$ is a production in G. Here, either x or y or both may be empty.

An item $[A \rightarrow x \,.\, y]$ is said to be *valid* for some viable prefix ux if and only if some right-most derivation

$$S \Rightarrow^* uAv \Rightarrow uxyv$$

exists. Note that xy is the handle of the sentential form uxyv, and that v is a terminal string. The mark "." in an item associated with some viable prefix indicates that this point in the production marks the end of the viable prefix.

There are in general many items valid for a given viable prefix, although the number must be finite. There can only be a finite number of items altogether, since they consist of a finite number of mark positions in a finite number of productions.

Now consider some state P in the finite state control for an LR parser. State P is associated with some set of viable prefixes, and therefore with some finite set of items valid for those viable prefixes. We propose to identify and distinguish the states in the finite control of the parser by constructing the set of valid items associated with each of the states.

For example, consider the machine of figure 6.1, and state 12. A list of some viable prefixes associated with state 12 is given above. We now construct a set of valid items associated with state 12.

Because state 12 is entered upon scanning "+", the item $[E \rightarrow E+ \,.\, T]$ must be associated with state 12. Although this is the only possible item in which the mark can follow "+", as required by the in-symbol of state 12, this is not the only item associated with state 12. The following are, too:

$[T \rightarrow .F]$
$[T \rightarrow .T*F]$
$[F \rightarrow .(E)]$
$[F \rightarrow .a]$

To show this association, note that the viable prefix

$$E+$$

is part of a sentential form

$$E+T$$

with the derivation

$$E \Rightarrow E+T \Rightarrow E+F$$

and therefore the item [T → .F] is valid for the viable prefix "E+". (From the definition given earlier, in this instance we have S = E, u = E+, x = ϵ, y = F, v = ϵ).

Similarly, the derivation

$$E \Rightarrow E+T \Rightarrow E+T*F$$

shows that the item [T → .T*F] is valid for the viable prefix "E+".

Item-Set Construction

We now see how a finite set of items and a (possibly) infinite set of viable prefixes may be associated with each state in the LR control automation. We next need to develop rules for building the item-sets and from them, the states and transitions of the LR automaton. The following rules do just that. We shall in due time show that they are correct and make each aspect of them apparent.

The three construction rules for an LR(0) control automaton are called the *start*, the *completion*, and the *read* operations, respectively, and are as follows:

The Start Operation. If S is the start symbol of the grammar, and S → w is some production, then item [S → .w] is associated with the start state.

This operation is needed to get the construction process started. The other three assume that some states and items associated with the states already exist. The start state may eventually have several items in it.

The Completion Operation. If [A → x . Xy] is an item in some state P, then every item of the form [X → .z] must be included in state P. Note that X must be a nonterminal symbol, and this rule must be repeated until no more new items can be added to the state.

The Read Operation. Let X be a terminal or nonterminal symbol in an item [A → x.Xy] associated with some state P. Then [A → xX.y] is associated with a state Q (possibly the same as P), and a transition

$$P \text{ to } Q \text{ on symbol } X$$

exists.

We call this a *read* operation, since it becomes a READ action in the final parser if X is a terminal symbol. When X is a nonterminal symbol, we have seen that a transition on X becomes part of the APPLY action, following the replacement of the handle on the stack top by that nonterminal symbol.

Parser Construction

These three operations are all that are needed to generate a finite control for an LR(0) parser, such as the one in figure 6.1, except for lookahead states. We shall see why lookahead states are necessary and how they are constructed shortly. For now, let us summarize the construction of the finite state system as implied by the three operations just given.

1. Give the start state a number, and use the start operation to put one item into it. Then use the completion operation repeatedly if necessary, to get more items into this state. In completing a state, we look for a nonterminal symbol X that follows the mark ".", and then add items of the form [X → .w] to the state, where X → w is a production. Eventually, this completion operation has to end.

2. We now have one state, consisting of a set of items. It will not be any different in principle than any other state constructed by this process, so we consider it a general state.

3. Use the read operation to start one or more new states, based on the present state. The idea is to look for items of the form [A → x.Xy], i.e., items in which some symbol X follows the mark, then build a new state from the item [A → xX.y], i.e., the mark "moved past" the symbol X. This new state incidentally must also contain all the other items from the old state in which this symbol X follows the mark. For example, if the old state contains the two items

[E → E+.T]
[E → .T]

(among others), then the new state must contain (at least) the items

[E → E+T.]
[E → T.]

Let the old state be P and the new state be Q. Then we know that the finite state machine in the LR parser will have the transition P to Q on X. This fact explains why we carry over into state Q all the items in which X follows the mark.

The "new" state may in fact be exactly like some other state previously constructed. We can tell whether two states are equivalent by examining the list of items associated with them. If the two lists are identical, when completed, then the two states must be equivalent. In this way, the state-building process must eventually terminate, since there can only be a finite number of distinguishable sets of items. Eventually, we have to construct a state that is identical to some state previously constructed.

It is not necessary to compare all the items in two states to judge their equivalence. We need compare only the *core* items; these are the items that are initially placed in a state through the read or start operation. Usually, they have some symbol preceding the mark, but they may not if empty productions exist in the grammar.

4. Complete the new state started in step 3 by applying the completion operation repeatedly.

5. Repeat steps 3 and 4 until no more new states are obtained.

An Example. Let us apply the construction to grammar G_0; the yield should be the finite-state machine in figure 6.1. We shall indulge in a bit of sleight-of-hand in assigning numbers to the states, since we know in advance that the first eight states are associated with completed items for the eight productions in G_0. There is no reason to number the states one particular way or another, except that we already have introduced the finite control for the parser (figure 6.1), and it is less confusing to adopt the state numbers used there.

The productions of G_0 are:

1. $G \rightarrow E\perp$
2. $E \rightarrow E+T$
3. $E \rightarrow T$
4. $T \rightarrow T*F$
5. $T \rightarrow F$
6. $F \rightarrow (E)$
7. $F \rightarrow a$

The production numbers 1 to 7 will be apply state numbers. We assign the next number, 8, to the start state, which initially contains the item $[G \rightarrow .E\perp]$. Now this item can be completed, by adding to state 8 all the items associated with E productions: $[E \rightarrow .E+T]$, $[E \rightarrow .T]$. The second of these in turn leads to a further completion operation, bringing the items $[T \rightarrow .T*F]$ and $[T \rightarrow .F]$ into state 8. Finally, the last item calls for a completion with the items $[F \rightarrow .(E)]$ and $[F \rightarrow .a]$. This operation yields the set of items in state 8, as follows:

8. $[G \rightarrow .E\perp]$
   $[E \rightarrow .E+T]$
   $[E \rightarrow .T]$
   $[T \rightarrow .T*F]$
   $[T \rightarrow .F]$
   $[F \rightarrow .(E)]$
   $[F \rightarrow .a]$

We now apply the read operation to these items. The first two items, with E following the mark, yield a new state, 9, and a transition (8 to 9 on E). The next two items yield state 10, and a transition (8 to 10 on T). The next item ([T → .F]) yields state 5 and a transition (8 to 5 on F). Finally the last two items yield states 11 and 7, with transitions on "(" and "a", respectively. Each of these states must be worked on by the completion and read rules, yielding new states, etc. Let us therefore continue with state 9.

State 9 arises from the two items [G → .E⊥] and [E → .E+T]. The read operation requires moving the mark past the E symbol, hence state 9 is initialized with the core set

9. [G → E.⊥]
   [E → E.+T]

This set is complete, since there are no nonterminal symbols following a mark in either item. The read operation applied to this set yields states 1 (for the first item), with the transition (9 to 1 on ⊥), and state 12 (for the second item), with the transition (9 to 12 on "+").

State 1 therefore stems from the first item in 9 and looks like this:

1. [G → E⊥.]

No completion or read operations are possible. This kind of item, with the mark at the end of the production, is called a *completed* item and is normally associated with a reduction, or apply state.

Another apply state obtained from state 8 is state 7:

7. [F → a.]

We next show the complete state set when the completion and read operations are carried out to the bitter end. We have added the state transitions to the right of the production, in the form (transition symbol, next state):

8.	[G → .E⊥]	(E, 9)
	[E → .E+T]	(E, 9)
	[E → .T]	(T, 10)
	[T → .T*F]	(T, 10)
	[T → .F]	(F, 5)
	[F → .(E)]	((, 11)
	[F → .a]	(a, 7)

9.	[G → E.⊥]	(⊥, 1)
	[E → E.+T]	(+, 12)
10.	[E → T.]	
	[T → T.*F]	(*, 13)
11.	[F → (.E)]	(E, 14)
	[E → .E+T]	(E, 14)
	[E → .T]	(T, 10)
	[T → .T*F]	(T, 10)
	[T → .F]	(F, 5)
	[F → .(E)]	((, 11)
	[F → .a]	(a, 7)
12.	[E → E+.T]	(T, 15)
	[T → .T*F]	(T, 15)
	[T → .F]	(F, 5)
	[F → .(E)]	((, 11)
	[F → .a]	(a, 7)
13.	[T → T*.F]	(F, 4)
	[F → .(E)]	((, 11)
	[F → .a]	(a, 7)
14.	[F → (E.)]	( ), 6)
	[E → E.+T]	(+, 12)
15.	[E → E+T.]	
	[T → T.*F]	(*, 13)

The first seven states correspond to the set of completed productions:

1. [G → E⊥.]
2. [E → E+T.]
3. [E → T.]
4. [T → T*F.]
5. [T → F.]
6. [F → (E).]
7. [F → a.]

We now have enough of the finite state controller for a comparison with figure 6.1. Only the state transitions and an indication of the parse states need survive.

Unfortunately, we have a problem with this grammar. Although we would like to assign state 2 to the item [E → E+T.], we see that it in fact appears with another item in state 15. Similarly, the item [E → T.] should be alone in state 3, but appears in state 10 with another non-completed item, [T → T.*F]. What are we to make of this situation?

Consider state 10, which contains the item [E → T.] and the item [T → T.*F]. When the finite control reaches this state on some sentential form, there may be two possible moves. The completed item [E → T.] essentially says that the parser should reduce the T on the stack top to E. On the other hand, the item [T → T.*F] essentially says, "Don't reduce the stack top, shift token * (if seen) into the stack". For example, the sentential form "T*a" leads to just such an indecision. We cannot have it both ways, and we don't want the parser to be nondeterministic, so something must be done to make a deterministic decision at this point.

Inadequate States

State 10 is called an *inadequate* or *inconsistent* state. In general an inadequate state is any state containing both a completed item (of the form [A → w.]) and any other item. Such a state represents a conflict in a parsing decision, in the same way that a conflict in an operator precedence table represents a potential difficulty in parsing some string. The conflict arises from the grammar. There are grammars for which no LR conflict arises, and others that produce one or more conflicts.

There is no easier way to determine in advance of an item-set construction whether a given grammar will create a conflict. We must simply forge ahead into constructing the state sets and discover the conflicts as they arise. The resolution of inadequate states will be discussed in section 6.1.4.

If the LR states contain no inadequate states, the grammar is said to be LR(0). We are then also sure that the grammar is unambiguous. We shall not prove this result, but it should be apparent from the deterministic character of the parsing machine that results from the state set construction. Of course, we have not yet shown that the parsing machine accepts exactly those sentential forms in the source grammar. We shall prove this assertion next.

Exercise

1. Construct the LR state sets for the following grammar G_3:

prog → block
block → BEGIN decls stmts END
block → stmt
decls → ϵ
decls → decls D ;
stmts → stmts ; stmt
stmts → stmt
stmt → block
stmt → S

Identify its inadequate states. Check the parser by tracing some strings in $L(G_3)$ and some strings not in $L(G_3)$.

Construction Correctness Proof

A proof of correctness of the parser system defined above will establish that the finite state machine constructed (hereafter simply called "the machine") recognizes exactly the class of viable prefixes of right-most sentential forms. The proof is in two parts. We first show that the items within any state are exactly consistent with the class of viable prefixes for that state. Then we can show that the machine recognizes viable prefixes exactly.

The machine may or may not contain inadequate states. The proofs essentially deal only with read and apply states, and even these are not distinguished as such. We merely consider a machine that can accept a left-most sentential form up to, but not past, its handle, i.e., a viable prefix.

We need a few definitions first.

Recall that an item [A→ x . y] is said to be *valid* for some viable prefix ux if and only if some right-most derivation

$$S \Rightarrow^* uAv \Rightarrow uxyv$$

exists.

A string w is said to be *associated with* or *valid for* some state P in the machine if and only if the machine falls into state P upon scanning w.

An item is said to be *valid* for some state P in the machine if and only if it is valid for some viable prefix w associated with state P.

Now we may state and prove the first part of the construction correctness.

Lemma 5.2. Every state P contains all and only the items valid for P.

Proof: "Only the" Part. Let [A → x.y] be in P through the construction process, and assume that it was placed in P through k + 1 steps of the process. We assume the lemma true for $k' \leq k$ steps.

If $x = \epsilon$, and $P \neq S$, then this item got into P through a completion operation through some other item, $[B \rightarrow .Az]$, also in P. By the inductive hypothesis, this item is valid for P, hence there must be a viable prefix u and a derivation

$$S \Rightarrow^* uBv \Rightarrow uAzv \Rightarrow uyzv$$

Therefore $[A \rightarrow .y]$ is also valid for P since viable prefix u is.

If $x = \epsilon$ and $P = S$, then k may be 1, so that we have item $[S \rightarrow .y]$, which is clearly valid for the start state. For $k > 1$ and $P = S$, we have item $[A \rightarrow .y]$, which is valid for the start state by an argument similar to that in the preceding paragraph.

For nonempty x, the item $[A \rightarrow x.y]$ has the form $[A \rightarrow rX.y]$, where $|X| = 1$. It can only be in state P because an item $[A \rightarrow r.Xy]$ exists in some other state R, with a transition (R to P on X). Now $[A \rightarrow r.Xy]$ is valid for some viable prefix ur, valid for R, by the inductive hypothesis, through a derivation

$$S \Rightarrow^* uAz \Rightarrow urXyz$$

But this also shows that $[A \rightarrow rX.y]$ is valid for the viable prefix urX. Finally, if ur is valid for state R, and a transition (R to P on X) exists, then urX is valid for P. Therefore $[A \rightarrow rX.y]$ is valid for P. QED

Proof: "All the" Part. Consider any viable prefix w valid for state P. We use induction on the length of w.

For $|w| = 0$, P must be the start state S_0. (There are no empty moves from S_0 to any other state). But S_0 contains at least one item $[S_0 \rightarrow .u]$ which is valid for the empty viable prefix.

Now let $|w| = k + 1$, and assume the lemma true for all strings w of length k or less. String w has the form

$$w = xX$$

where $|X| = 1$ and X is in $(N \cup \Sigma)$. With w valid for P, we have the derivation

$$S \Rightarrow^* wy = xXy$$

Consider the step in which X was introduced through a production:

$$S \Rightarrow^* rAz \Rightarrow rsXtz \Rightarrow^* rsXy$$

where $rs = x$, $tz \Rightarrow^* y$, and $A \rightarrow sXt$. This derivation shows that $[A \rightarrow s.Xt]$ is valid for $w = xX = rsX$, hence for P. Is this item in P? Note that $[A \rightarrow s.Xt]$ is valid for viable prefix x; since xX is valid for P, then there must be a state R for which x is valid, with a transition (R to P on X). However, the inductive

hypothesis implies [A → s.Xt] must be in R, therefore by the construction, [A → sX.t] is in P. We have shown that every state contains all the items valid for it. QED

The Main Theorem

We now turn to the principal theorem:

Theorem 5.5. The machine M (constructed as above) recognizes exactly the class of viable prefixes of grammar G.

An immediate consequence of this theorem is that M will serve as a (possibly) nondeterministic bottom-up parser for sentences in G. It will be deterministic if there are no inadequate states; in that case, the handle of any sentential form will be identified by falling into an apply state containing a single completed item [A → w.]. If there are inadequate states, then each of these represents a nondeterminism which must be resolved by lookahead methods to be developed later.

Proof: (M recognizes every viable prefix in G.) First, let w be some viable prefix in G; then we have

$$S \Rightarrow^* wv \text{ (right-most)}$$

Let v be terminal, without loss of generality. We prove by induction on the number of derivation steps that M recognizes w.

For zero steps, the start state clearly recognizes the class of empty viable prefixes. Therefore, consider a (k + 1)-step derivation,

$$S \Rightarrow^* uAv' \Rightarrow uxv'$$

where A is such that w is some prefix of ux. If w is a prefix of u, then by the inductive hypothesis, M recognizes u and therefore w. Hence let w be a prefix of ux but not u. Let $x = X_1X_2 \ldots X_ny$, where each of the X_i are of length 1, such that

$$w = u\, X_1\, X_2 \ldots X_m$$

and $m \leq n$. We then have

$$S \Rightarrow^* uAv' \Rightarrow u\, X_1\, X_2 \ldots X_n\, y\, v'$$

Now u is accepted by M by falling into some state P_0 for which u is valid, by induction. The derivation shows that [A → .X_1 X_2 ... X_n y] is valid for u, and by lemma 5.2, must be in P_0. Then by the construction process, there must be a sequence of states P_1, P_2, ... P_n associated with items and viable prefixes as follows:

- P_1 with $[A \rightarrow X_1.X_2X_3 \ldots X_n\ y]$ and uX_1.
- P_2 with $[A \rightarrow X_1X_2.X_3 \ldots X_n y]$ and uX_1X_2.

.
.
.

- P_n with $[A \rightarrow X_1X_2X_3 \ldots X_n.y]$ and $uX_1X_2 \ldots X_n$.

But this construction also shows that M accepts $uX_1X_2 \ldots X_m = w$. QED

Proof: (M recognizes only viable prefixes in G.) Let w be a string accepted by M. By induction on $|w|$, we have:

For $|w| = 0$, M accepts w in the start state, and the empty string is a viable prefix in every grammar.

For $|w| = k+1$, consider the next-to-last move of M, from some state R to P on some symbol X:

$$w = rX$$

By induction, r is a viable prefix valid for R. Also because of the transition (R to P on X), and lemma 5.2, every item valid for R is in R. There must therefore be an item of the form $[A \rightarrow x.Xy]$ in R such that a derivation

$$S \Rightarrow^* uAz \Rightarrow uxXyz$$

exists, with $ux = r$. But this derivation also shows that $uxX = w$ is a viable prefix and therefore valid for state P. QED

Finiteness of Machine States

We have not required that the number of states in M be finite in these proofs, only that they be countable (induction requires the countable property). The proofs establish a correspondence between the states, the set of items constructed for them, and the (usually infinite) set of viable prefixes in the grammar. However, the state set must be finite, since there are a finite number of possible ways of constructing item sets. We have therefore also indirectly shown that the set of viable prefixes in some grammar are recognized by a finite state automaton.

6.1.4. Resolution of Inadequate States

We now return to the problem of the inadequate states that can arise in the construction of an LR parsing machine. We can deal with these in one of several different ways, as follows:

1. Discard the LR(0) construction and attempt the construction of an LR(k) machine, first for $k=1$, then $k=2$, etc. We shall define the construction of an LR(k) machine later. We note here only that the size of each state set and the number of states increases exponentially with k, so that the construction is practical only for small k or a small grammar.

2. A full LR(k) construction can be compromised in the interests of reducing the number of states and state-set size by merging certain states. The result, if successful, is identification of the lookahead set for every inadequate state, as well as generation of the LR machine. This technique yields a so-called *LALR(k)* parser. We shall discuss LR(k) construction in section 6.2.

3. The LR(0) machine itself can be used to infer lookahead sets for the inadequate states. A common algorithm used for the purpose is given next. Essentially, we may infer k-token lookaheads by backtracking the LR(0) machine from an inadequate state. We call such a resolution an *SLALR(k)* machine.

4. The lookaheads may be determined by computing the FOLLOW sets for the nonterminal left member of the production in the inadequate state. When this is done, we have an *SLR(k)* resolution.

These resolution approaches are ranked in order of decreasing strength. Thus there are grammars that yield to an SLALR treatment, but not an LALR, etc. Nevertheless, even the SLR(k) resolution approach is sufficiently powerful to be useful for practical grammars and is the easiest to implement.

SLR stands for "simple LR", and was first reported by DeRemer [1971]. *LALR* stands for "lookahead LR", and is described in Aho and Johnson [1974].

We defer the two LR(k) resolutions (for $k>0$) until a later section, as it requires considerable development. We must first introduce a more general definition of a state-set item and a more general construction system.

SLALR(1) Resolution

The inadequate states may be resolved by computing a set of lookaheads associated with transitions in the LR(0) machine, using the machine transitions themselves. Although weaker than a merged-state LR(k) parser, it is more powerful than an SLR(k) resolution. As usual, we consider only $k=1$.

1. Let P be some inadequate state in an LR(0) machine. It therefore contains at least one completed item in addition to some others of any kind.

2. For each completed item in P, create a new state P′ and a set of lookahead transitions (as yet unlabeled with symbols) from P to P′. Associate the completed item with P′, removing it from P. A lookahead transition is similar to a read transition, except that the read head is not moved.

Figure 6.5 illustrates steps 1 and 2. State 12 has two read transitions, on T and on F, to states 15 and 17, respectively. It is also associated with the completed items [F → .] and [E → ET.]. Then steps 1 and 2 result in figure 6.5(b), in which new states 12′ and 12″ are created to accept the completed items. The labels L1 and L2 represent a set of lookahead symbols (yet to be determined) for the new states.

We now have only apply states, each containing a single completed item, and read states, each possessing only read or lookahead transitions to other states. Although this machine is not exactly the same as that shown in figure 6.1, it may readily be transformed into it.

We next associate a *lookahead set*, consisting of terminal symbols, with each transition as follows.

3. The lookahead set for a terminal read transition is the symbol associated with the transition.

4. Consider a nonterminal read transition T to some state P. Its lookahead set is the union of the lookahead sets into P (if P is an apply state) or the transitions out of P (if P is a read state).

5. Let R be an apply state with production A → w, and let Q be a state with a lookahead transition Q′ to R. Let P(R) be the set of read states in M such

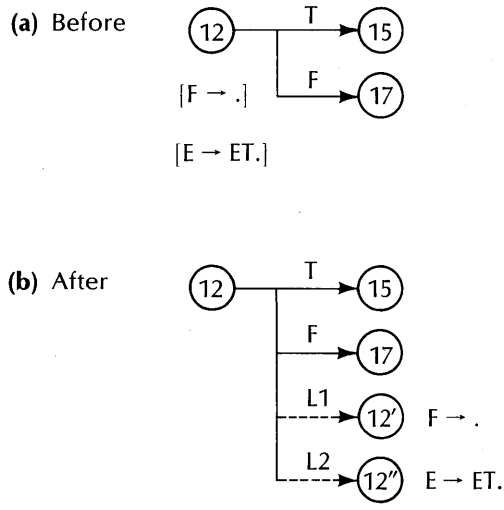


Figure 6.5. Splitting an inadequate state.

that for each X in P(R), configuration (X, w) $\vdash^*$ (Q, ϵ) $\vdash$ (R, ϵ). That is, we find all the states X such that the machine can accept w by starting from X and ending in R, by passing through Q.

Now there must be a transition T from X on the nonterminal symbol A (A is part of the production associated with R). This transition is guaranteed by the construction process. Add the lookahead set L(T) associated with T to the set L(Q'), for every X.

6. Steps 4 and 5 are repeated until no additions to any lookahead set can be made anywhere in the machine. At that point, the lookahead transitions into the apply state carry the desired SLALR(1) lookahead sets.

Example Construction. The following grammar generates an LR(0) machine with several inadequate states:

1. S $\rightarrow$ E$\perp$
2. E $\rightarrow$ E;T
3. E $\rightarrow$ T
4. T $\rightarrow$ Ta
5. T $\rightarrow$ ϵ

The LR machine for this grammar is shown in figure 6.6. It contains four inadequate states, 6, 8, 9, and 10. Lookahead transitions to apply states have been added in the figure, as demanded by steps 1 and 2 in the SLALR(1) algorithm.

Steps 4 and 5 are repeatedly applied to yield the decorated machine of figure 6.7. A suitable starting point is the E transition (6,7). By step 4, this transition has the lookahead set {$\perp$;}, from the transitions from state 7. Similarly, the T transitions (6,8) and (9,10) both include the set {a}. Eventually, these transitions will also pick up the lookahead sets from the transitions (10,2) and (8,3), but they are empty so far.

Step 5 applied to state 2 causes this state to pick up the symbols associated with the E transition (6,7), since only state 6 (through 7, 9, 10, 2) leads to state 2 on the string "E;T". These symbols {$\perp$;} are then associated with the transition (10,2), and also with transition (9,10).

States 5 and 5' are both associated with the production T $\rightarrow$ ϵ, and eventually become a single state in the final machine. However, the transitions into them are distinct. It can happen that the trace-back caused by step 5 can yield different lookahead sets for the two states, and therefore different lookahead sets for the transitions. The same state may therefore be accessed by different lookahead transitions.

We find that this grammar is SLALR(1). The test is whether the lookahead sets and the terminal read sets from each state are pair-wise disjoint, and they are for this machine. The lookahead sets we developed for the nonterminal transitions are not considered in the test.

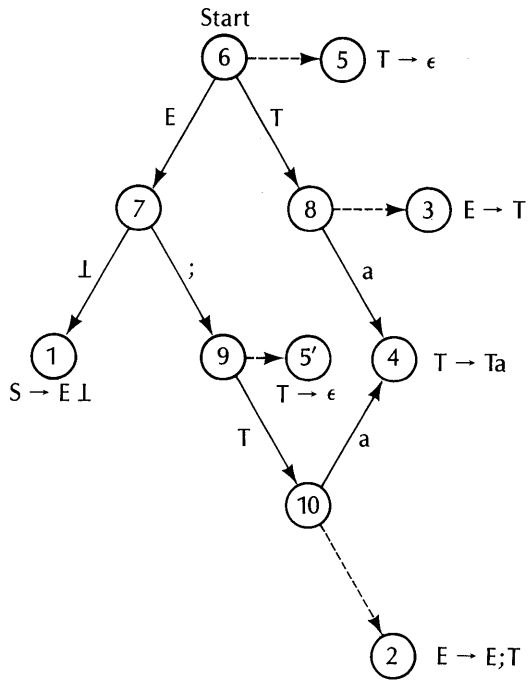


Figure 6.6. LR machine for an example grammar. The grammar productions are attached to states in the machine.

SLR(1) Resolution

Recall that a state P is inadequate if it contains a completed item $[A \rightarrow w.]$ and some other item, of any kind. The presence of $[A \rightarrow w.]$ in a state P means that there is some set of derivations of the form

$$S \Rightarrow^* xAy \Rightarrow xwy$$

such that the LR(0) machine accepts xw by falling into state P. Clearly, the set $L = FIRST_1(y\ FOLLOW_1(S))$ is the desired lookahead set for this item in P. However, L is a subset of $FOLLOW_1(A)$, clearly. The latter set is easy to determine, and if the use of FOLLOW is sufficient to resolve every inadequate state (it often is), then we have an SLR(1) resolution. A resolution of inadequate states in an LR(0) machine using $FOLLOW_k$ would be an SLR(k) resolution.

For example, consider the machine of figure 6.1, with the items developed as above. State 10 is inadequate, since it contains the two items

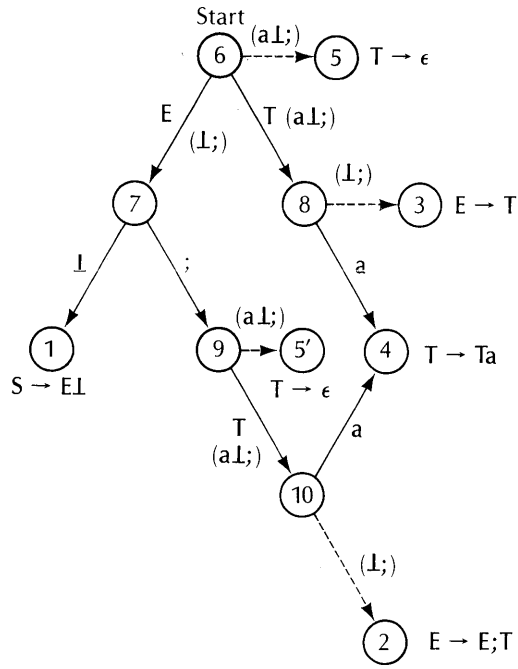


Figure 6.7. Machine of figure 6.6 decorated with lookahead sets.

[E → T.]
[T → T.*F]

The set FOLLOW(E) is {), +, ⊥}, as is easily shown from the grammar. (See chapter 4 for a complete discussion.) This set does not contain symbol "*", hence state 10 may be resolved by a one symbol lookahead. It is shown in figure 6.1 as a new state, 16, with two lookahead transitions to states 10 and 3.

Similarly, state 15, which contains the items

[E → E+T.]
[T → T.*F]

may also be resolved with the same FOLLOW(E) set. It is shown in figure 6.1 as a new state, 17, with lookahead transitions to states 2 and 15.

The use of the FOLLOW function may or may not resolve an inadequate state. If it fails, the grammar may still be SLALR(1), LALR(1) or LR(1). Essentially, the FOLLOW function yields all the symbols that can follow some nonterminal A in a sentential form. It makes no distinctions based on the particular inadequate state that we are interested in. The FOLLOW

function therefore tends to yield a larger set of lookahead symbols in general than can actually exist for that nonterminal in that state.

Exercises

1. Consider the grammar G_3

   $S' \rightarrow S$
   $S \rightarrow Aa \mid dAb \mid dca \mid cb$
   $A \rightarrow c$

   of Aho and Ullman (Aho [1972]). Show that it is SLALR(1) but not SLR(1).

2. Devise an efficient data structure in Pascal for a set of productions and the item-sets. For example, an item may be characterized by the number pair (i, j), where i is a production number and j is the mark position within the pair. What else might be added for the sake of an efficient LR(0) state set determination? Sketch the operations required for a general purpose parser generator.

6.2. LR(k) Parsers

We now describe a more general means of constructing an LR(k) parser and also introduce the concept of canonical tables and a canonical parser. These are useful in obtaining several important reductions in parser tables and are also referred to in many papers on LR(k) systems.

An *LR(k) item* is a marked production and a terminal string w in Σ^*. The length $|w|$ is at most k, and w is called a *lookahead string*. We denote such an item by:

$$[A \rightarrow x.y, w]$$

The marked production $A \rightarrow x.y$ is called the *core* of the item. Several items with the same core may be combined through the notation

$$[A \rightarrow x.y, \{u_1, u_2, \ldots u_n\}]$$

which stands for the set of items

$[A \rightarrow x.y, u_1]$
$[A \rightarrow x.y, u_2]$
.
.
.
$[A \rightarrow x.y, u_n]$

We are interested in a subset of all possible items. In particular, we will be constructing sets of items in such a way that for each item of the form $[A \rightarrow x.y, u]$, there exists some right-most derivation of the form

$$S \Rightarrow^* rAst \Rightarrow rxyst \Rightarrow^* r'x'ut$$

where

$$rx \Rightarrow^* r'x' \qquad \text{and}$$

$$ys \Rightarrow^* u$$

As a rationale for this form of item, consider a bottom-up LR(k) parser with rxy in the stack and ut next in the input list. String u is clearly a k-symbol lookahead, and among the productions that are potential candidates for the parse so far, $A \rightarrow xy$ is certainly one. If this production is next to be reduced, the lookahead string u, along with the current state, will indicate production $A \rightarrow xy$.

We see that by carrying along a lookahead string in the items when the parser state set is constructed, we will automatically have the necessary lookahead strings available. A k-symbol lookahead will be used on both read and reduce actions, for the sake of generality. This lookahead will always appear in an item-set as the lookahead string. It can be shown that an LR(k) constructor accepts the broadest class of grammars, larger than the SLR(k) or SLALR(k) constructors described earlier.

The LR(k) item-set construction proceeds in three steps as follows:

1. *Initial state construction*:

   (a) If $S \rightarrow w$ is a production, then add $[S \rightarrow .w, \epsilon]$ to the initial state. Symbol S is the grammar's start symbol.

   (b) If $[A \rightarrow .Bx, w]$ is in the initial state, and $B \rightarrow y$ is a production, then add all the items of the form $[B \rightarrow .y, z]$ to the initial state, where z is in the set $FIRST_k(xw)$. Of course, this is added to the set only if it is not already there. We repeat this step until no more items can be added to the initial state.

2. *Starting a new state*: Let [A → r.Xy, w] be an item in some state P, where X is a terminal or nonterminal token. We locate every item in state P such that X follows the mark ".", then start a new state Q with these items, but with the mark moved past X. Thus item [A → r.Xy, w] will become item [A → rX.y, w] in state Q. The set of items so used to start state Q is called the *core set* of Q.

   The core set of Q may be identical to the core set of some other state Q′ already constructed. If so, states Q and Q′ are equivalent and may be merged. By merging such states, we guarantee that the construction process will terminate in a finite number of steps.

   A transition from P to Q on X will be part of the LR(k) parser, thanks to this operation.

3. *Closing a new state*: For every item of the form [A → x.By, w] in a state Q, and for every production B → z, we add to Q all the items of the form [B → .z, u], where u is in the set $FIRST_k(yw)$. This step is repeated until no more items can be added to Q. After step 1 is finished, steps 2 and 3 are repeated alternately, in that order, until no more new states can be constructed or closed.

As before, some states will carry completed items, of the form [A → x., w]. For them, a reduction is indicated in that state, with the production A → x. Now there will in general be several items in that state of the form [A → x., ...]. We therefore have a set of lookahead strings W such that the reduction is permitted only if the next k (or less) tokens in the input list is among the set W. The lookaheads may also be used to resolve one of several possible reductions or to resolve a conflict between a reduction and one or more read operations.

It can be shown that the grammar is LR(k) if and only if the following condition is met by the state set constructed as above:

For every state P containing an item of the form [A → x., w], there exists no other item of the form [B → y.z, w] in P.

6.2.1. Canonical LR(1) Parsing Tables

For the sake of ease of comprehension, we specialize k to 1 in most of the following arguments. The extension to k > 1 is relatively straightforward, but tends to clutter up the notation.

A *canonical LR(1) parser* is a pair of functions f and g representing a finite control for a push-down automaton, as follows:

Function f. The *action function f* maps a pair {T, Y}, where T is a table name and Y is a member of $\Sigma \cup \{\epsilon\}$ into one of the following:

- APPLY n, where n is a production number.
- SHIFT, abbreviated "S".
- ACCEPT, i.e., "halt", abbreviated "A".
- ERROR, abbreviated "X".

The APPLY action will appear in the table as the production number. Thus an action table entry will be a production number, S, A, or X. We shall later introduce another entry, "@", that will mean *inaccessible.*

Each table T is associated with one state P generated in the LR(1) construction process. Then for every completed item of the form [A → x., w] in state P, we set f(T, w) = n, where n is the number of the production A → x. For every item of the form [B → u.xv, w] in state P, where |x| = 1, and x is a terminal token, we set f(T, w) = S, i.e., *shift.* If item [G → x., w] appears in state P, where G is the grammar's goal symbol, we set f(T,w) = A, i.e., *accept.* All other f entries for table T are set to X, i.e., *error* (although not all these can necessarily be reached with some input string).

Function g. The *goto function g* maps a pair {T, Y}, where T is a table name and Y is a member of $N \cup \Sigma \cup \{\epsilon\}$ into

- A table name T′ or
- ERROR, abbreviated "X".

Now table T is associated with some state P in the construction process. For every item of the form [A → u.xv, w] in that state, where |x| = 1, we set g(T, w) = T′, where table T′ is associated with a state Q containing the item [A → ux.v, w], and such that a transition from P to Q on symbol x exists. Note that such items cause *shift* to be entered in the appropriate column of the f function, if x is terminal. Here, however, x may be terminal or nonterminal. All other g entries in table T are ERROR.

6.2.2. A Canonical Parser

An LR(1) parser based on the functions f and g consists of a finite control (represented by the following Pascal program) and a stack containing table names. Let T0 be an initial table that corresponds to the initial state constructed. The parser may then be described by the following program. The functions in the program are explained through comments.

```
var X: char;      {an input token}
```

```
START: PUSH(T0);   {push the initial table name in the stack}
       X:=NEXTTOKEN;
          {variable X gets the next token}
  while ~ f(TOS, X) = "A"  do
     {TOS is the table on the top of the stack.  We continue
       this algorithm until the accept condition is found}
  begin  {accept state not reached if we enter this block}
     if f(TOS, X) = "S"  then
     begin  {a shift operation is wanted}
        PUSH(g(TOS,X));  {push the table indicated by the
                          goto table, based on token X}
        X:=NEXTTOKEN;    {get the next token}
     end
   else
     if f(TOS, X) = reduce n  then   {n is a production number}
     begin  {a reduce operation is wanted}
        APPLY(n);   {indicate to the rest of the compiler that
                     production n is next to be applied}
        POP(RIGHTMEMLEN(n));
                    {pop as many elements from the
                    stack as tokens in the right member of
                    production n}
        PUSH(g(TOS, LEFTMEM(n)))
                    {some table name was exposed
                    through the previous POP operation; this is
                    used in the g function, along with the
                    nonterminal left member associated with
                    production n, to push another table name}
     end
   else ERROR  {syntax error}
  end
```

The simplicity of this parser speaks for itself. We see from the reduce operation that the g function must contain mappings associated with nonterminal tokens in general. The shift operation implies that a g-mapping associated with terminal tokens is needed. When the end of the input list is reached, there may be several reduce moves left or some error indication; we therefore see that the action function f must include the empty string ϵ in its domain. Note that this parser does not require a special termination token.

Example. Let us construct an LR(1) canonical parser for grammar G_0, as follows:

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

We need the $FIRST_1$ sets for the nonterminals, as follows:

nonterminal v	$FIRST_1(v)$
E	(a
T	(a
F	(a

The initial state 0 contains the items $[E\rightarrow.E+T, \epsilon]$ and $[E\rightarrow.T, \epsilon]$, through construction rule 1(a). The completion rule on item $[E\rightarrow.E+T, \epsilon]$ implies that each of the following items must be included in state 0:

$[E\rightarrow.E+T,+]$, from $FIRST_1(+T)$
$[E\rightarrow.T,+]$, from production $E\rightarrow T$
$[T\rightarrow.T*F,+]$, from production $T\rightarrow T*F$
$[T\rightarrow.F,+]$, from production $T\rightarrow F$
$[T\rightarrow.T*F,*]$, from item $[T\rightarrow.T*F,+]$ and $FIRST_1(*F+)$

Sixteen items eventually appear in state 0. This state and the remaining states in the complete LR(1) state set are given below. The core items are marked ">".

State	Item	Lookaheads	Go to state
0	> $E\rightarrow.E+T$	ϵ +	1
	> $E\rightarrow.T$	ϵ +	2
	$T\rightarrow.T*F$	ϵ + *	2
	$T\rightarrow.F$	ϵ + *	3
	$F\rightarrow.(E)$	ϵ + *	5
	$F\rightarrow.a$	ϵ + *	4
1	> $E\rightarrow E.+T$	ϵ +	6
2	> $E\rightarrow T.$	ϵ +	reduce (2)
	> $T\rightarrow T.*F$	ϵ + *	7

3	>	T→F.	ϵ	+	*	reduce (4)
4	>	F→a.	ϵ	+	*	reduce (6)
5	>	F→(.E)	ϵ	+	*	8
		E→.E+T	)	+		8
		E→.T	)	+		9
		T→.T*F	)	+	*	9
		T→.F	)	+	*	10
		F→.(E)	)	+	*	12
		F→.a	)	+	*	11
6	>	E→E+.T	ϵ	+		13
		T→.T*F	ϵ	+	*	13
		T→.F	ϵ	+	*	3
		F→.(E)	ϵ	+	*	5
		F→.a	ϵ	+	*	4
7	>	T→T*.F	ϵ	+	*	14
		F→.(E)	ϵ	+	*	5
		F→.a	ϵ	+	*	4
8	>	F→(E.)	ϵ	+	*	15
	>	E→E.+T	)	+		16
9	>	E→T.	)	+		reduce (2)
	>	T→T.*F	)	+	*	17
10	>	T→F.	)	+	*	reduce (4)
11	>	F→a.	)	+	*	reduce (6)
12	>	F→(.E)	)	+	*	18
		E→.E+T	)	+		18
		E→.T	)	+		9
		T→.T*F	)	+	*	9
		T→.F	)	+	*	10
		F→.(E)	)	+	*	12
		F→.a	)	+	*	11
13	>	E→E+T.	ϵ	+		reduce (1)
	>	T→T.*F	ϵ	+	*	7
14	>	T→T*F.	ϵ	+	*	reduce (3)

15	>	F→(E).	ϵ + *		reduce (5)
16	>	E→E+.T	) +		19
		T→.T*F	) + *		19
		T→.F	) + *		10
		F→.(E)	) + *		12
		F→.a	) + *		11
17	>	T→T*.F	) + *		20
		F→.(E)	) + *		12
		F→.a	) + *		11
18	>	F→(E.)	) + *		21
	>	E→E.+T	) +		16
19	>	E→E+T.	) +		reduce (1)
	>	T→T.*F	) + *		17
20	>	T→T*F.	) + *		reduce (3)
21	>	F→(E).	) + *		reduce (5)

The canonical tables found from these state-sets are as follows:

	action, f						goto, g							
	a	+	*	(	)	ϵ	E	T	F	a	+	*	(	)
0	S	X	X	S	X	X	1	2	3	4	X	X	5	X
1	X	S	X	X	X	A	X	X	X	X	6	X	X	X
2	X	2	S	X	X	2	X	X	X	X	X	7	X	X
3	X	4	4	X	X	4	X	X	X	X	X	X	X	X
4	X	6	6	X	X	6	X	X	X	X	X	X	X	X
5	S	X	X	S	X	X	8	9	10	11	X	X	12	X
6	S	X	X	S	X	X	X	13	3	4	X	X	5	X
7	S	X	X	S	X	X	X	X	14	4	X	X	5	X
8	X	S	X	X	S	X	X	X	X	X	16	X	X	15
9	X	2	S	X	2	X	X	X	X	X	X	17	X	X
10	X	4	4	X	4	X	X	X	X	X	X	X	X	X
11	X	6	6	X	6	X	X	X	X	X	X	X	X	X
12	S	X	X	S	X	X	18	9	10	11	X	X	12	X
13	X	1	S	X	X	1	X	X	X	X	X	7	X	X

	action, f						goto, g							
	a	+	*	(	)	ϵ	E	T	F	a	+	*	(	)
14	X	3	3	X	X	3	X	X	X	X	X	X	X	X
15	X	5	5	X	X	5	X	X	X	X	X	X	X	X
16	S	X	X	S	X	X	X	19	10	11	X	X	12	X
17	S	X	X	S	X	X	X	X	20	11	X	X	12	X
18	X	S	X	X	S	X	X	X	X	X	16	X	X	21
19	X	1	S	X	1	X	X	X	X	X	X	17	X	X
20	X	3	3	X	3	X	X	X	X	X	X	X	X	X
21	X	5	5	X	5	X	X	X	X	X	X	X	X	X

The following is a trace of the LR(1) parser for the string "a+a*a":

stack	X (next token)	input	action	
0	a	+a*a	S	
0,4	+	a*a	6	F→a
0,3	+	a*a	4	T→F
0,2	+	a*a	2	E→T
0,1	+	a*a	S	
0,1,6	a	*a	S	
0,1,6,4	*	a	6	F→a
0,1,6,3	*	a	4	T→F
0,1,6,13	*	a	S	
0,1,6,13,7	a	ϵ	S	
0,1,6,13,7,4	ϵ		6	F→a
0,1,6,13,7,14	ϵ		3	T→T*F
0,1,6,13	ϵ		1	E→E+T
0,1	ϵ		A	(accept)

6.2.3. Table Reductions

We shall now explore several kinds of reductions that may be made on a set of canonical LR(1) tables. (These reductions also apply to canonical LR(k) tables for k>1. An LR(k) table maps a string of length k to reduce, shift, etc.)

The following discussion will be based on the canonical parser given earlier in this section.

The general strategy is to first identify those entries that are inaccessible. They will be some of the "X" entries, which will be marked "@". Since these entries can never be reached on scanning any input string (whether in the language or not), they can in fact be made anything at all. We can then redefine them in an attempt to reduce the number of tables.

One reduction can be made through *error postponement*. Here we look for X entries in the f table such that, when replaced by some *reduce*, permits the merger of two or more states without affecting the detection of the error ultimately. Essentially, we allow some more spurious reduce operations, rather than report the syntax error immediately, but require that at some future point in the parse the error associated with the next token will be detected and reported. We will not allow some sequence of reduce operations to end in a state that can accept the next error token.

Another reduction is possible through the removal of transitions connected to single productions. A single production has the form A→B, where B is a nonterminal. Usually, single productions require no action on the part of the rest of the compiler; for example, in building the abstract syntax tree, those nodes associated with a production A → B need not be added. The result of removing state transitions associated with single productions is often a sizable reduction in the canonical table, and a concomitant improvement in parser speed.

Each of these reductions will in general yield some new canonical table set containing inaccessible states or pairs of equivalent states. The inaccessible states may be discarded and the equivalent states merged. When the reduced canonical table is obtained, it may easily be transformed into a set of packed sparse matrix tables, an example of which is given in figure 6.3.

Identification of Inaccessible Entries

Here, we wish to identify every X entry in a canonical table that can never be reached.

We first demand that a syntax error be detected as early as possible. The canonical parser algorithm clearly does this by inspecting f(T, h), where h is the next input token and T is the current table (or state). If f(T, h) is neither accept, shift nor reduce, then it reports an error. We say that token h is an *error token*; the string w preceding h is such that there exists a set of strings wy in the language accepted by the parser, but h is not $\text{FIRST}_1(y)$ for any y.

Since the parser will always first detect an error in the f table, it follows that none of the X entries in the g table are accessible, and each of these may therefore be changed to "@".

Now let us examine the X entries in the f table. Let T be a table and h some token, such that f(T, h) = X. The parser clearly will be able to access f(T, h) only if one of these conditions holds: (1) T is the initial table T0, or (2) there is some table T′ and token h′ such that f(T′, h′) = *shift* and g(T′, h′) = T.

Case (1) applies upon first entering the parser. Table T is the initial table, and the algorithm must be prepared to report error on any possible first token.

Case (2) says that f(T, h) can be accessed if table T can be reached after

some shift operation. If table T can only be reached after a reduce operation, and the next token is an error token, the error will be detected before the reduce operation is begun, through some X entry in the f table.

We therefore keep every X entry in the T_0 table. We then follow every shift entry in the action table through its goto table; if a shift action ends in a transfer to table T, then we must keep every X entry in the action part of table T.

Note that the action part of a table T will carry X entries or "@" entries; no mixture of the two is possible.

Example. Consider the tables for G_0 given previously. All the X entries in the goto table may be changed to "@". All the X entries in the action table T0 must be retained. However, all the X entries in the action tables 1,2,3,8,9,10,13,14,18, 19, and 20 may be changed to "@", since none of these tables is the target of a shift action. For example, table 1 appears in only one place in g, as g(0,E); it is not the target of a shift. Similarly, table 2 only appears in g(0,T). On the other hand, table 5 appears as g(0,"("), and f(0,"(")=shift, so table 5 must contain X entries.

Postponement of Error Checking

Consider tables 15 and 21 for G_0:

	action, f						goto, g							
	a	+	*	(	)	ϵ	E	T	F	a	+	*	(	)
15	X	5	5	X	X	5	X	X	X	X	X	X	X	X
21	X	5	5	X	5	X	X	X	X	X	X	X	X	X

These two tables are clearly equivalent, except for the ")" and "ϵ" columns in the action part. Suppose we changed these into the following tables:

	action, f						goto, g							
	a	+	*	(	)	ϵ	E	T	F	a	+	*	(	)
15	X	5	5	X	5	5	X	X	X	X	X	X	X	X
21	X	5	5	X	5	5	X	X	X	X	X	X	X	X

We have changed the parser so that in certain circumstances, some reduce operations will be permitted that otherwise would be considered an error. Eventually, however, the error must be detected, and must not lead to an "@" in an f-table. We must therefore examine the consequences of each such change of X to *reduce n.*

Let us therefore consider a set of tables S_T that are equivalent except for some X entries in the action part that we would like to change into reduce operations. We need an algorithm that will test whether we can safely do this, and that will indicate other necessary changes (e.g. "@" entries that must become X). Note first that X and "@" entries in the set S_T may be considered the same; we can always change "@" entries back to X as needed. In what follows, assume that every such change is made; "@"'s are changed to X's as needed, and some X's in f are changed to reduces. (We must later retract all this, if the test for its permissibility is negative.)

Consider one such table entry in table T, with an entry f(T, h)=X that we would like to change to *reduce n*, where production n is A→w. In the reduction, |w| table names are dropped from the stack, exposing some table T′. We must next consider g(T′, h). If this is "@", we can change the "@" to X, to catch the error there. However, g(T′, h) may be some table T″. We must then examine f(T″, h); this must either be *error* or another reduce, e.g., *reduce m*. If the latter, the search must continue deeper. If the search leads to some table T_3 such that f(T_3, h)=*shift*, then the change of f(T, h) from X to *reduce n* cannot be made.

Further development of this reduction algorithm will lead to a difficult combinatorial problem, the optimal selection of subsets of tables that can be made equivalent under this transformation. Suppose that we find that some subset S′ is equivalent under error postponement. Its equivalence is in general the result of some modifications elsewhere in the table (some changes of "@" to X), and they must be preserved upon the next equivalence consideration. The problem is that one set of tables S′ may not be equivalent unless some other set S″ is similarly shown to be equivalent, or a set S′cannot be equivalent if S″ is made equivalent. We see that in general, an optimal reduction can only be made if all possible combinations are tested with the objective of finding the least final table set.

A nearly optimal approach is the following:

Step 1. Identify those subsets of tables that are candidates for an equivalence reduction. These must necessarily be disjoint. Order the subsets in order of decreasing size, and apply step 2 to them in that order.

Step 2. Given a subset, make the necessary changes in its tables for an error postponement equivalence and test the validity of the transformation. Make whatever other changes are necessary. Each of these changes must be revocable. If the test is positive, fix the changes, merge the states, and apply step 2 to the next state subset. If the test is negative, revoke all the changes and apply step 2 to the next state subset.

In this algorithm, we have no assurance that an optimal solution will be found. It is conceivable that a merger of a set of three states will preclude

mergers of several subsequent sets of two states each.

The NEXT Function

A critical step in this algorithm is the identification of that set of tables that can appear just below some handle in the stack. Let this set be designated NEXT(T, i), where T is a top-of-stack table and i is a production in f(T, h) for some token h. Let production i be $A \rightarrow w$, where $w = w_1 w_2 ... w_n$. Then table T'will be in NEXT(T, i) if and only if a goto sequence of the following kind exists:

$$
\begin{aligned}
g(T', w_1) &= T_1; \\
g(T_1, w_2) &= T_2; \\
&\vdots \\
g(T_n, w_n) &= T.
\end{aligned}
$$

This sequence follows from the property of an LR parser that its finite-state machine (represented by the goto tables) will accept the viable prefix of a right-most sentential form, and the correspondence of the tables (or states) on the stack to the terminal and nonterminal symbols of the viable prefix.

Now the error postponement algorithm can be given in more detail:

1. Identify a set of tables P that are candidates for error postponement. These tables will agree, except for conflicts between "@", X, and *reduce n*.

2. Among the tables P, change each @ to X and each X to *reduce n* as needed in order that they all agree. Let production n be $A \rightarrow w$. Then there must not exist a path in the goto table from T0 to T on a suffix of w; all paths from the initial table must either spell w fully, or spell some string w′ whose suffix is w. This condition guarantees that the stack will contain enough states to permit a reduction. For every change of X to a *reduce n*, perform step 3.

3. Given a change of some X to a *reduce n* in f(T,h) (table T, token h), then repeat step 4 for every table T′ in NEXT(T,n), where n is the production number in step 2 above.

4. Given a table T′ and token h from step 3. The value of g(T′, h) may be a table T″, "@", or X. If "@", change it to X to catch the error. f(T′, h) will be either "@", X, *shift*, or *reduce m*. If "@", change "@" to X. If *shift*, the set P cannot be merged; halt. If *reduce m*, table T′ may be in set P and m = n; if so, ignore it, otherwise set P cannot be merged; halt.

If the algorithm halts agreeably in step 4, then the set P can be merged as indicated in step 2.

Example. Consider tables 15 and 21 in the canonical parser for G_0. We are interested in changing f(15, ")") to 5 and f(21,"ϵ") to 5. Production 5 is F→(E), and an examination of the g table shows that tables 0, 6, and 7 will lead to f(15, ")") on the handle "(E)"; NEXT(15, 5)={0,6,7}. In every case, f(T',")")=X, where T' is in NEXT(15, 5), so this appears safe. (No "@" entry in g need be changed, incidentally).

Next consider changing f(21,ϵ) to 5. (The ϵ means that the parser has reached the end of the input list). Now NEXT(21,5)={5, 12, 16, 17} from the g table. Again, in every case, f(T',ϵ)=X, and no "@" entry in a g table need be changed. We therefore conclude that tables 15 and 21 may be safely merged through error postponement.

Removal of Single Production Transitions

Consider the single production E → T in grammar G_0, and a derivation

$$E \Rightarrow T \Rightarrow T*F$$

for example. Since the reduce step associated with E ⇒ T rarely (if ever) is needed as a semantics action step, we would like to transform the canonical parsing tables such that the derivation is effectively

$$E \Rightarrow T*F$$

The result is fewer parser operations and a smaller table.

Typical programming language grammars contain many such productions. They generally appear in the definition of the expression operators, and there are as many single productions as there are precedence hierarchy levels in the language. A parenthesized expression often calls for a long sequence of single production reduction steps. We would clearly like to be able to remove as many single production reduction steps as possible.

Suppose, then, that the action corresponding to a production A→B is to be eliminated from the canonical tables; we want no reduce step on this production to ever appear. Any pair of steps of the form

$$xAy \Rightarrow xBy \Rightarrow xwy$$

where B→w is some production is to be effectively replaced by

$$xAy \Rightarrow xwy$$

during parsing.

We examine every table T' containing a *reduce p* action, where production

p is A→B. Let T″ be a table in NEXT(T′,p). Then table T″ can be removed only if, for all tokens h:

{ 1. f(T′,h) = f(T″,h)
or 2. One of f(T′,h), f(T″,h) is @
or 3. f(T′,h) = *reduce p*}

and

{ 4. g(T′,h) = g(T″, h)
or 5. One of g(T′,h), g(T″,h) is @ }

If this condition holds, then we can eliminate table T″ by setting f(T′, h):=f(T″, h) if f(T′, h)=@, and by setting g(T′, h):=g(T″, h) if g(T′, h)=@. Then every reference to T″ can be changed into a reference to T′, throughout the table set.

Note: (1) When T″ is removed from the table, NEXT will be affected in general. It must be recomputed for each such reduction. (2) The algorithm will fail if some nonterminal derives only the empty string. (3) The algorithm will fail if some single production A→C, where C is not B, also exists.

Condition (2) is rather stringent; useful programming languages often contain one or more nonterminals that only derive the empty string. (They are useful as place markers for certain semantic steps). Condition (3) is less likely.

The removal of these restrictions requires a more elaborate algorithm. Soisalon-Soininen [1977] gives a bibliography of work on this problem and a general solution.

Equivalence

Consider a set of canonical tables, possibly containing @ entries, which may have been subjected to one or more of the above reductions. We may then perform an equivalence reduction along the lines described in chapter 3. Let the set of canonical tables be partitioned into their k-equivalence classes (k is an arbitrary integer, not the "k" of LR(k)). We then refine the partition to yield a (k+1)-equivalence partition as follows:

Let tables T and T′ belong to one of the k-equivalence sets P. Then T and T′ belong to the same (k+1)-equivalence set if, for every token h:

{ f(T, h) = f(T′, h), or one of f(T, h), f(T′, h) = @}

and

{ g(T, h) and g(T′, h) are both in P, or one of g(T, h), g(T′, h) = @}

Refinement should be repeated until no proper refinement is obtained. Then each partition represents an equivalence class of tables. Each such class may be merged as follows:

Let T = T′, i.e., both belong to the same equivalence class. We eliminate T′, retaining T, as follows:

1. If f(T, h) = @ for some token h, then set f(T, h) := f(T′, h). Note that f(T′, h) may also be @.
2. If g(T, h) = @ for some token h, then set g(T, h) := g(T′, h).
3. Replace T′ by T everywhere in the canonical tables.
4. Delete table T′.

The initial partition might as well be the single set consisting of the complete set of tables. The first pass through the equivalence reduction will usually yield many 1-equivalent sets, and a typical grammar will require no more than two or three passes.

Exercises

1. Construct a set of LR(1) canonical tables for the grammar

   $G \rightarrow S$
   $S \rightarrow b\ D$
   $S \rightarrow E\ e$
   $D \rightarrow D\ d$
   $D \rightarrow \epsilon$
   $E \rightarrow E\ s$
   $E \rightarrow s$
   $E \rightarrow S$

   Identify the inaccessible table entries, identify states that can be merged through error postponement, states that can be merged by removing the parsing actions associated with the two single productions, and finally identify the equivalent state sets.

2. Develop an algorithm that accepts an LR(1) canonical table set and yields READ, LOOKAHEAD, APPLY, and PUSH tables, with "else" default columns in the LOOKAHEAD and APPLY tables.

3. Write a Pascal program that constructs a set of LR(1) canonical tables. For practical grammars, a large number of LR(1) tables will be generated. Develop efficient means of storing and manipulating the state sets through a mass storage medium (disk or drum).

6.2.4. LALR(1) Tables

A set of LALR(1) tables may be constructed from the set of LR(1) tables as follows:

Recall that an LR(k) item has the form

$$[A \rightarrow x.y, w]$$

where $A \rightarrow xy$ is a production and w is a lookahead string, consisting of terminal tokens. The portion $A \rightarrow x.y$ of the item is called the *core* of the item. Now two items are said to belong to the same LALR(k) parser state if they have the same core. We essentially ignore the lookahead string w in forming a state. In doing so, we arrive at the same state set as in an LR(0) construction, but the items will carry lookahead set information derived from the LR(k) construction process. The lookahead sets may then be used to resolve any inadequate states.

For example, consider the LR(1) construction for grammar G_0 given previously. Its states may be merged as follows to yield a set of LALR(1) states:

(0), (1), (2, 9), (3, 10), (4, 11), (5, 12), (6, 16), (7, 17), (8, 18), (13, 19), (14, 20), (15, 21).

The resulting FSA is isomorphic to the LR(0) machine given in section 6.1. It has two inadequate states (ignoring the lookaheads), the merged states (2,9) and (13,19). However, these are easily resolved through the LR(1) lookahead strings carried into the merger. Thus the merged state (2,9) has the following set of items:

(2,9)	> [E→T., ϵ +)]	reduce (2)
	> [T→T.*F, ϵ + *]	7

The second item clearly applies on a lookahead set {*}, since token "*" follows the mark. The first item applies on a lookahead set {ϵ, +,)}. Since the two lookahead sets are disjoint, the state is resolvable.

An LALR(k) resolution is more powerful in general than either an SLR(k) or an SLALR(k) resolution; i.e., there exist grammars that are resolvable by LALR(k), but not SLALR(k) or SLR(k), and there exists no grammar resolvable by either SLALR(k) or SLR(k) and not LALR(k).

As a practical matter, the construction of an LR(1) item set requires a very large memory, compared to an LR(0) item set. Most practical LALR(1) constructors therefore merge core-items as they are generated. Unfortunately, the lookaheads are also merged and the construction is thereby weakened. Pager [1977] discusses this problem in depth and gives a general merging algorithm that preserves as much lookahead information as possible,

without demanding as much storage capacity as an LR(1) constructor would. In Pager's algorithm, the question of state inadequacy is examined when the merger of a pair of states is being considered. The merger is allowed if an inadequacy is not thereby created and disallowed otherwise. Now such a state merger in general requires further work in that state and in all other states that have been derived from it; Pager's algorithm deals with the problem of fixing up the items that must change in response to the merger. The method is complicated and the reader is referred to his paper for further details.

6.3. Augmented Grammars

An alternative way of generating an LR(0) finite-state machine is through an *augmented grammar*, derived from a context-free grammar G, by the following algorithm:

1. Suppose there are n productions in the grammar. We then create a new set of symbols not already in the grammar, #i, where $1 \leq i \leq n$. We thus have a unique symbol for each production.
2. Create a set of new symbols that correspond to the nonterminals in G, i.e., if A is a nonterminal in G, then create a new symbol A′.
3. Construct a new regular grammar G′ from G, as follows:

   (a) For every production $A \rightarrow w$ in G, with index i, add $A' \rightarrow w\ \#i$ to grammar G′. (A′ corresponds to A).

   (b) For every nonterminal B in a right-part of a production $A \rightarrow xBy$, add production $A' \rightarrow x\ B'$ to grammar G′. (A′ corresponds to A, and B′ to B.)
4. The nonterminals of G′ are the symbols A′, B′, ... that correspond to the nonterminals A, B, ... of G.
5. The terminals of G′ are all the remaining symbols—the unmarked nonterminals of G, the terminals of G, and the special symbols #i.
6. Augment G with a special start symbol S″ and a production $S'' \rightarrow S'$.

We now clearly have a regular grammar. Each of the productions are of the form

$A \rightarrow x$ where x is a terminal string, or
$A \rightarrow yB$ where y is a terminal string or empty, and B is a nonterminal symbol.

A finite-state machine can clearly be constructed for this grammar, by

methods described in chapter 3. Furthermore, any FSA can be transformed into a deterministic FSA. What properties does the deterministic FSA have? Some reflection will reveal that:

1. The states will correspond to subsets of the nonterminals in G′.
2. The start state will correspond to state S″.
3. The halt states will be entered upon scanning one of the special #i symbols, since only these productions in the regular grammar end in a terminal symbol.
4. Given some sentence in the language L(G′), the machine will of course accept the sentence by falling into a halt state upon scanning the sentence.
5. Now the sentences of L(G′) are viable prefixes of sentential forms in G. (We shall prove this shortly.) Therefore our FSA appears to be exactly the LR machine that we wish—it will scan a right-most sentential form uxv, with x the handle, and will find a #i transition precisely upon scanning the viable prefix ux; the #i will indicate the correct production to apply. However, there is no guarantee that #i will be the only transition from the state reached upon scanning ux. If there is another transition, then we contend the grammar G is not LR(0); the grammar G is LR(0) if and only if for every state S with an out-going "#" transition, there are no other out-going transitions.

Example Consider grammar G_0. The augmented grammar G_0' is given below:

$S'' \rightarrow E'$
$E' \rightarrow E + T \ \#1$
$E' \rightarrow E + T'$
$E' \rightarrow E'$ {trivial productions may be dropped}
$E' \rightarrow T \ \#2$
$E' \rightarrow T'$
$T' \rightarrow T + F \ \#3$
$T' \rightarrow T + F'$
$T' \rightarrow F \ \#4$
$T' \rightarrow F'$
$F' \rightarrow (E) \ \#5$
$F' \rightarrow (E'$
$F' \rightarrow a \ \#6$

The FSA given in figure 6.1 (aside from the lookahead states) may be derived from this regular grammar (left as an exercise).

Let us now show that the viable prefix of a sentential form (augmented by a #i symbol) is derivable in G′. As an example, consider the right-most sentential form

$$E + (E) * a$$

in G_0. The viable prefix "E + (E) #5" is derivable in G′ as follows:

$$S'' \Rightarrow E' \Rightarrow E + T' \Rightarrow E + F' \Rightarrow E + (E) \#5$$

The FSA based on G′ will move through a sequence of states associated with the string "E+(E)", and the next state will then carry a transition on symbol #5. In figure 6.1, the state sequence is 8, 9, 12, 11, 14, 6. State 6 is associated with production number 5, F → (E), and will therefore carry an out-going transition on symbol #5 in the FSA derived from G′.

We may compare the derivation of E+(E)*a in G with the G′ derivation given above:

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+(E)*a$$

Note that the second and third steps are bypassed in the G′derivation through the production T′ → F′.

Correspondence of G to G′

We now state and prove the theorem we asserted:

Theorem 5.1. A string "u x #i" is derivable in grammar G′ if and only if "uxv" is a right-most sentential form in grammar G, and x is its handle.

Proof: "If" Part. We prove this part by induction on the number of derivation steps for uxv. For 0 steps, the sentential form in G is S, and S′ is trivially derivable from S″ in G′.

For n steps, consider the last step:

$$uAv \Rightarrow uxv$$

By induction, the handle of uAv (wherever it is), augmented by some #i, is derivable in G′. Now the handle of uAv cannot lie entirely in u, since if it did, it would reduce to a nonterminal B to the left of A, violating the right-most derivation condition. Therefore the handle of uAv must include A or lie to right of A. (Note that the handle may be empty; the preceding sentence is true nevertheless). We then have two cases, as follows:

CASE 1.

$$u' B v' \Rightarrow u' r A s v' = u A v \Rightarrow u x v \quad \text{in G.}$$

Here the handle of uAv includes A. Let production B→rAs be number j. By induction, we have

$$S'' \Rightarrow^* u' B' \Rightarrow u' r A s \ \#j \text{ in } G'$$

But then we also have the production $B' \rightarrow rA'$ in G', yielding

$$S'' \Rightarrow^* u' B' \Rightarrow u' r A' \Rightarrow u' r x \ \#i$$

where A→x is production number i.

CASE 2.

$$u A r B v' \Rightarrow u A r s v' = u A v \Rightarrow u x v$$

Here the handle of uxv is in v. Again, let production B→s be number j and production A→x be number i. By induction we have

$$S'' \Rightarrow^* u A r B' \Rightarrow u A r s \ \#j \qquad \text{in } G'$$

In this derivation, there had to be a step that introduced A through a production

$$C' \rightarrow g A h$$

i.e.,

$$S'' \Rightarrow^* u' g A h h' \qquad \text{in } G'$$

By the nature of the derivations in G', h' must be empty and h contains a nonterminal in G'; the string u'gA is therefore uA. We also have an alternative production

$$C' \rightarrow g A'$$

in G', hence

$$S'' \Rightarrow^* u A' \Rightarrow u x \ \#i.$$

QED
Proof of the only if part is left to an exercise.

LR(k) Augmented Grammar

Knuth [1965] develops a similar algorithm for the general case LR(k). Essentially, the nonterminals of G' are of the form [A; x] where A is a nonterminal in G, $|x| = k$, and x is terminal and derivable from the string yv in a right-most sentential form uyv, where

$$S \Rightarrow^* uAv \Rightarrow uyv$$

The productions in G are then transformed into productions in G'through

the use of these special nonterminals and also through the special symbols #i, in a manner similar to the above.

The grammar so obtained is a regular grammar, and can be shown to derive viable prefixes, together with the k-symbol lookahead. The FSA accepts a viable prefix and k symbols past it, then is in a state with a #i out-directed transition, where i indicates the production associated with the handle of the viable prefix.

Exercise

Write an augmented grammar for the context-free grammar

$$
\begin{aligned}
G &\rightarrow S \\
S &\rightarrow b\ D \\
S &\rightarrow E\ e \\
D &\rightarrow D\ d \\
D &\rightarrow \epsilon \\
E &\rightarrow E\ s \\
E &\rightarrow s \\
E &\rightarrow S
\end{aligned}
$$

then construct a reduced finite state machine from the augmented regular grammar.

6.4. Size of LR(k) Tables

Purdom [1974] gives a number of statistical formulas that yield the size of an LALR(1) parser reasonably accurately, without having to construct the parser. His study is based on 84 grammars, 65 of which had 15 or less terminals and nonterminals and the remaining 19 of which contained more than 15.

He found that the number of states S is a linear function of the grammar size G, as follows

$$S = bG + a$$

where

$$a = .02 \pm .45$$

and

$$b = .5949 \pm .0048$$

The number of transitions T is a linear function of a sum Q:

$$T = b'Q + a',$$

where

$$a' = 3.4 \pm 2.9$$

and

$$b' = .9867 \pm .0069$$

The small tolerances indicate that the number of states and transitions can be rather accurately estimated from the grammar. The grammar size G is the sum of the lengths (in tokens) of the right sides of the productions, plus the number of productions. Thus G is 18 for grammar G_0.

The sum Q is more complicated to determine. We begin with the grammar, and augment each production with the special token #i, where i is the production number. Then we represent the grammar as a set of trees. Each tree is defined on a nonterminal and contains all the augmented productions for that nonterminal. Any two productions of the forms

$$A \rightarrow xy$$
$$A \rightarrow xz$$

are represented by a common path from the root A that spells out the common prefix x. The suffixes y and z then are separate subtrees rooted in the right-most token of x. Thus, for the augmented grammar

$$S \rightarrow E \perp \#1$$
$$E \rightarrow E + T \#2$$
$$E \rightarrow T \#3$$
$$T \rightarrow F \uparrow T \#4$$
$$T \rightarrow F \#5$$
$$F \rightarrow a \#6$$
$$F \rightarrow (E) \#7$$

we have the trees shown in figure 6.8. Note that the tree edges are labeled with the production right-side symbols. The nodes are labeled with integer counts, determined as follows.

The count associated with a tree node is the size of the set $\{FIRST_1(A_1), FIRST_1(A_2), \ldots FIRST_1(A_n)\}$, where the transitions out of node N are labeled $A_1, A_2, \ldots, A_n$. Thus in figure 6.8, the root node of the E tree has cardinality 5, since the union of the sets $FIRST_1(E)$ and $FIRST_1(T)$ is the set {E, T, F, a, (}, of cardinality 5. The leaf nodes of the trees carry count 0, since they have no children.

Finally, the count Q is the sum of all the node counts for the production trees, less the counts for the root nodes, plus the count for the start root. Thus

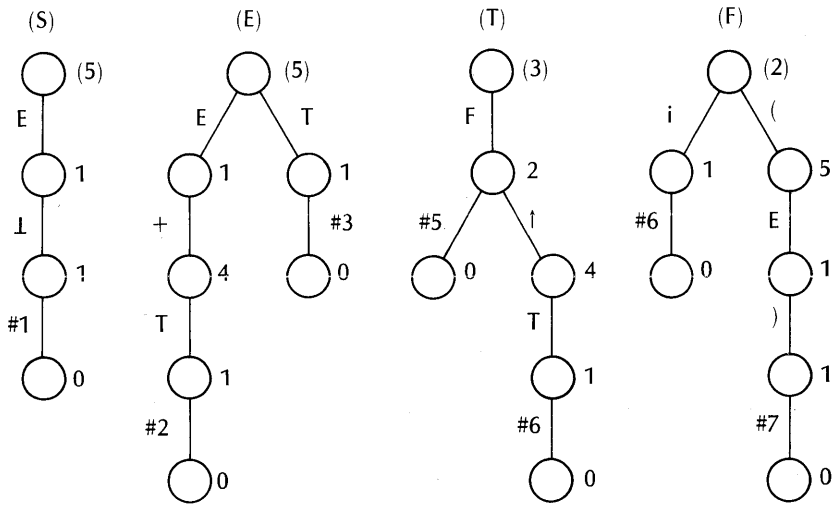


Figure 6.8. Production trees for Purdom's parser transition estimate.

we count only the start node root and all the remaining nonroot nodes. In figure 6.8, Q = 29.

Purdom's estimate for the number of transitions in this grammar is therefore T = 31.0. (The actual number of transitions is 30.) The estimated number of states S = 12.51, and the actual number is 13.

6.5. Comparison of Parsing Methods

The various parsing methods developed in this and the preceding chapters have important differences that have a direct bearing on the ease of construction of a viable production set, the construction of suitable semantics, the memory required for the parser, and the execution speed of the parser.

Generality

The LR(k) grammars and languages cover both the LL(k) and precedence grammars and languages. Thus, given an arbitrary grammar, the chances are greater that a LR(k) parser can be constructed for it than any other. Of LL(k) and the precedence grammars, little can be said. There are LL(1) grammars that are not simple (1,1) precedence and vice versa.

If we consider the class of extended grammars and their relation to recursive descent parsers, we find that a relatively straightforward extended grammar can be constructed for every language described by an LR(1) simple

grammar. Indeed, all the common programming languages have been implemented by recursive descent compilers, and such a compiler may be based on an extended grammar as we have seen. However, it is not clear that the LR(1) languages are equivalent to the 1-symbol extended grammar, recursive descent languages.

Recall that a non-LL(1) grammar can often be transformed into an LL(1) grammar. The practical consequence of such transformations is that an LL(1) parser has been constructed for several common languages, e.g., Algol (Lewis and Rosenkrantz [1971]). However, the transformed grammar invariably contains many more productions than an equivalent LR(1) grammar. The transformation is also not amenable to an algorithmic method, so that errors can be introduced into a given source grammar.

Of all the source grammar representations, the most easily understood is probably the extended grammar. Each right part is a regular expression and has a FSA representation that corresponds to a syntax graph of the language. One therefore has a very direct relationship between a very commonly used form of language definition and the basis grammar for an automatic parser generator.

Parser Size

Each of the parsers can be represented as a set of interpreted tables. Of all the parsing methods, the LALR parser tables are invariably smallest. For example, LaLonde [1971] gives the following comparative table sizes (in bytes) for four languages, and three different parsers: MSP, mixed-strategy precedence; WWSP, Wirth-Weber simple precedence; and LALR:

Grammar	Productions	MSP bytes	WWSP bytes	LALR bytes
XPL	109	3274	*	1250
EULER	120	3922	4321	1606
EULER-II	100	3017	3204	1276
ALGOL60	173	>6800†	>6100*	2821

* Not a WWSP grammar
† Not a MSP grammar

Comparative studies of LL(1) and LALR(1) parsers are not available, although there is reason to believe that they require comparable table space. The net space required by a recursive descent parser is, in our experience, about twice that required for an equivalent LALR parser.

Empirical Speed Comparisons

The comparison of parsing speed is subject to many variables only some of which depend directly on the parsing algorithm. Most modern computer systems employ a virtual memory, code segmentation, data segmentation, or some combination thereof. In such a system, only certain portions of the data or code is in fast memory at any instant; the bulk of the data and code is in auxiliary slow memory, e.g., a disk or drum. The execution speed of any program therefore depends critically upon such factors as (1) the amount of fast memory available, (2) the efficiency of the memory manager, (3) the access time of disk, and (4) the *working set* of the algorithm, that is, the amount of code and data that should be resident in fast memory for high performance.

An LALR or LL table-driven parser tends to require a rather large working set. It is difficult to choose a subset of the tables and code that would permit the parsing of a section of a program; in general, all the tables and the entire parsing algorithm are needed many times during the parsing of almost any source statement.

On the other hand, the working set of a recursive descent parser (not table driven) tends to be relatively small. The parsing of some statement may invoke relatively few procedures out of the entire set comprising the parser. For example, those procedures designed to parse the declarations would never be called while parsing the executable statements.

The semantic operations of a recursive descent compiler are often part of the parsing procedures. This practice tends to keep the working set small during compilation. The semantic operations of a table-directed parser must be separate from the tables, which means that the tables are of no value during code generation and vice versa.

Unfortunately, an empirical study of these matters has never been reported. Although the memory requirement for the LALR tables is smaller than for a recursive descent parser, it is by no means clear that an LALR-based compiler will be superior in overall performance.

LaLonde reports some performance comparisons for an MSP and an LALR parser for several grammars, as follows:

Program	Records	Tokens	Reductions	MSP secs	LALR secs
Compactify	77	439	1,262	0.84	0.52
LCS	3,322	17,369	58,707	28.86	15.88
XCOM	4,241	24,390	66,108	45.35	25.11
DOSYS	7,291	29,334	81,581	55.58	30.49

These results indicate a substantially greater efficiency for the LALR

parser. It is not immediately obvious just why this should be. The MSP parser uses fast indexing into a precedence matrix, based on two tokens, while the LALR parser must scan a list of tokens determined by its READ state. However, upon a reduction decision, the MSP parser must then scan a table of production right parts, comparing them with the stack contents, in order to determine the production, while the LALR parser yields the next production as part of the same function that locates the end of the handle. This latter saving apparently outweighs the cost of searching a READ state table.

6.6. Bibliographical Notes

The LR(k) parsing system is generally attributed to Knuth [1965], in which both the item-set construction and the augmented grammar construction are presented, with arguments for their correctness. He also proved that the LR(k) grammars are the largest class that can be deterministically parsed left-to-right with a k-symbol lookahead. Despite the significance of these results, very little additional work on LR parsers was done until more recently.

The SLR(k) resolution is from DeRemer [1969] and [1971]. The SLALR(k) resolution was first reported by LaLonde [1971a]. The LALR(k) system was first proposed by Anderson, Eve, and Horning (Anderson [1973]); it and the SLR(k) system seem to be the LR systems of choice for the known LR parser generators.

The optimizations of canonical LR(k) tables are from Aho [1972].

CHAPTER 7

SYNTAX-DIRECTED TRANSLATION

We are conceptually at the midpoint of our subject. We have developed mechanisms for recognizing the structure of sentences written in the source programming language. The end result of this process of lexical analysis and parsing is a sequence of productions that constitute a bottom-up or top-down parse of the input. We may also view the end result (so far) as a derivation tree.

We have seen how a source string can be subdivided and analyzed according to an underlying context-free grammar and transformed into a structure (a derivation tree), in terms of which the underlying meaning of an input sentence can be defined.

We must now develop tools for the construction of the translation or object sentence. The output of any compiler can be viewed as a sentence in some object or target language, whether the object be another high-level programming language, symbolic assembly language, or sequences of binary machine instructions.

The general task of accepting the tree structure created by a parser and generating from it the object sentence (or code) is called the *synthesis* or *semantics* phase of a compiler. The synthesis phase can be conceptually subdivided into a number of reasonably distinct subtasks, as described in chapter 1, namely: collection and distribution of attributes of the data objects found in the abstract-syntax tree, optimization on the tree, allocation of data object space, and code generation.

7.1. General Principles

A *syntax-directed translation scheme*, or *SDTS*, specifies a source language, a target language, and rules for the translation of any string in the source language into a target string. The rules of an SDTS are production rules, extended to include translation forms. No mechanism for the translation of source to target is implied by the SDTS definitions, although we shall see that the translation can easily be achieved.

As a conceptual model, the SDTS provides a good framework for understanding some of the underlying principles of translation. An implementation of an SDTS is a *string translator*, but of a limited kind. The SDTS model does not contain provisions for collecting and distributing attributes, and without attributes a translator is of little practical use. Nevertheless, certain ordering properties stem from the SDTS model, and these are of considerable value in planning an attribute system for a complete compiler.

We shall first define the SDTS model and explore some of its properties. A general implementation will be presented. The chapter ends with the development of a general purpose translator system that can serve as the framework of a practical bottom-up compiler.

7.1.1. Definitions

A *syntax-directed translation scheme*, is a 5-tuple consisting of

1. A finite *input alphabet* Σ, comprising the symbols of the source language.
2. A finite set N of *nonterminal symbols* used in a context-free grammar to define the input language.
3. A finite *output alphabet* Δ, containing symbols that will appear in the *translation* or *output string.*
4. A finite set of *rules* R of the form $A \rightarrow w, y$, to be defined later.
5. A *start symbol* S in N with the same meaning and use as in the definition of a context-free grammar.

A rule in R of the form

$$A \rightarrow w, y$$

is such that "w" consists of a string of terminals and nonterminals, just as in a context-free grammar, and "y" is a string of symbols from N and Σ. The symbol A is in N. Furthermore, there must be a one-to-one association of nonterminals in "w" with the nonterminals in "y". The string "w" is called the *source element* and the string "y" is called the *translation element* of this rule.

The 4-tuple (N, Σ, P, S), where P is a set of rules of the form $A \rightarrow w$, and $A \rightarrow w, y$ is a rule in R, is called the *underlying source grammar* of T. That is, we obtain the underlying source grammar by stripping the translation element from the rules and discarding the output alphabet.

There is also an underlying *target grammar*, obtained by removing the source element from each production and discarding the input alphabet.

For example, consider the SDTS $S_1 = (\{E, T, A\}, \{a, b, c, +, -, [,]\}, \{ADD, SUB, NEG, x, y, z\}, R, E)$, where R consists of the set of translation rules

1. $E \rightarrow E + T,\ T\ E\ ADD$
2. $E \rightarrow E - T,\ E\ T\ SUB$
3. $E \rightarrow - T,\ T\ NEG$
4. $E \rightarrow T,\ T$
5. $T \rightarrow [\ E\ ],\ E$
6. $T \rightarrow A,\ A$
7. $A \rightarrow a,\ x$
8. $A \rightarrow b,\ y$
9. $A \rightarrow c,\ z$

The underlying source grammar of S_1 is

$$E \rightarrow E+T \mid E-T \mid -T \mid T$$
$$T \rightarrow [E] \mid A$$
$$A \rightarrow a \mid b \mid c$$

A *translation form* is a pair of strings (u, v), such that "u" is a sentential form of the underlying grammar of an SDTS and "v" is a *translation* consisting of elements drawn from N and Σ.

A translation form is defined as follows:

1. (S, S) is a translation form, and the two S's are said to be *associated*. Also, S is the start symbol of the SDTS.
2. If (a A b, a′ A b′) is a translation form, and the two A's are associated; further if $A \rightarrow g, g'$ is a rule in R, then (a g b, a′ g′ b′) is also a translation form. The association of nonterminals in g to those in g′ must be carried into the translation form exactly as it is in the rule.

The notation

$$(a\ A\ b,\ a'\ A\ b') \Rightarrow (a\ g\ b,\ a'\ g'\ b')$$

expresses the transformation of one translation form into another.

We see that the first part of the translation form is exactly a sentential form of the underlying CFG grammar; the second part is an associated translation, a sentential form of the underlying target grammar.

The *translation* defined by an SDTS T, is the set of pairs

$$\{(x, y) \mid (S, S) \Rightarrow^* (x, y),\ x \in \Sigma^*,\ \text{and } y \in \Delta^*\}$$

which is clearly analogous to the definition of a language in a context-free grammar given in chapter 2.

Example. Consider the input string

$$-[a+c]-b$$

that is derivable in the underlying grammar of S_1 as follows:

$$\begin{aligned} E &\Rightarrow ET \Rightarrow -T-T \Rightarrow -[E]-T \Rightarrow -[E+T]-T \\ &\Rightarrow -[T+T]-T \Rightarrow -[A+T]-T \Rightarrow -[a+T]-T \\ &\Rightarrow^+ -[a+c]-b \end{aligned}$$

The derivation tree for this string is shown in figure 7.1(a).

Now the translation for this string is given below. We have introduced subscripts on certain nonterminals in the translation forms as needed to indicate the association between the input and output nonterminals. Thus two T's appear in the second translation form; they are distinguished by the subscripts 1 and 2.

$$\begin{aligned} (E, E) &\Rightarrow (E-T, E\ T\ SUB) \\ &\Rightarrow (-T_1-T_2, T_1\ NEG\ T_2\ SUB) \\ &\Rightarrow (-[E]-T, E\ NEG\ T\ SUB) \\ &\Rightarrow (-[E+T_1]-T_2, T_1\ E\ ADD\ NEG\ T_2\ SUB) \\ &\Rightarrow^+ (-[A+T_1]-T_2, T_1\ A\ ADD\ NEG\ T_2\ SUB) \\ &\Rightarrow (-[a+T_1]-T_2, T_1\ x\ ADD\ NEG\ T_2\ SUB) \\ &\Rightarrow^+ (-[a+c]-T, z\ x\ ADD\ NEG\ T\ SUB) \\ &\Rightarrow^+ (-[a+c]-b, z\ x\ ADD\ NEG\ y\ SUB) \end{aligned}$$

The translation of "$-[a+c]-b$" in the SDTS T is therefore:

z x ADD NEG y SUB

We evidently have a translator for some simple arithmetic expressions to a postfix notation, with the twist that pairs of addition operands are interchanged in the output string.

We have introduced some lexical operations in this grammar, through the last three translation rules: $A \rightarrow a, x \mid b, y \mid c, z$, in order to clarify the translation process. Of course, a set of identifiers will appear as one terminal symbol in a practical compiler and be distinguished by the lexical analyzer.

7.1.2. Tree Transformations

A tree interpretation of the syntax-directed translation process is shown in figure 7.1. Part (a) of this figure shows the derivation tree for the string "$-[a+c]-b$" in the underlying grammar of T. The translation can be viewed as a transformation of this tree into another tree, the transformation consisting of (1) removing the terminal nodes, (2) permuting the children of each interior node according to the appropriate translation rule, and (3) adding terminal nodes that correspond to the translation terminal set Δ.

Thus figure 7.1.(b) shows the derivation tree in (a) with the terminal nodes removed. In this step, it is possible that two different productions can appear to be the same. For example, the productions

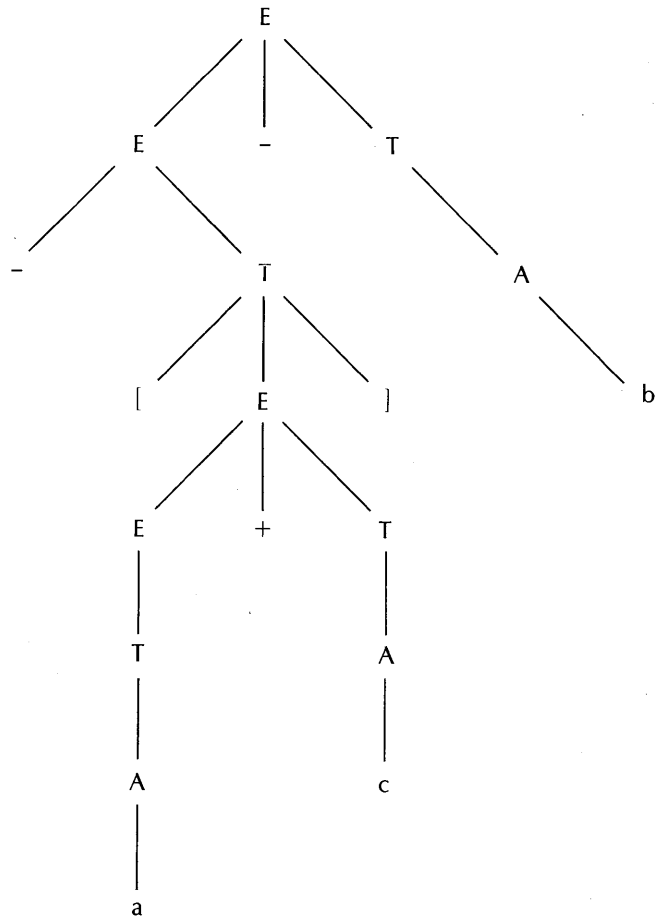


Figure 7.1(a). Derivation tree for $-[a+c]-b$.

$$E \rightarrow -T \quad \text{and}$$
$$E \rightarrow T$$

look the same in the tree after the terminal nodes are removed. We therefore label each node with a production number.

In figure 7.1(c), the children of each node have been permuted according to the translation rules and the terminal output symbols have been added. Note that the children of each interior node now are exactly a translation element of some translation rule. For example, the children of node $_1$E were originally

$$E + T$$

The translation rule is

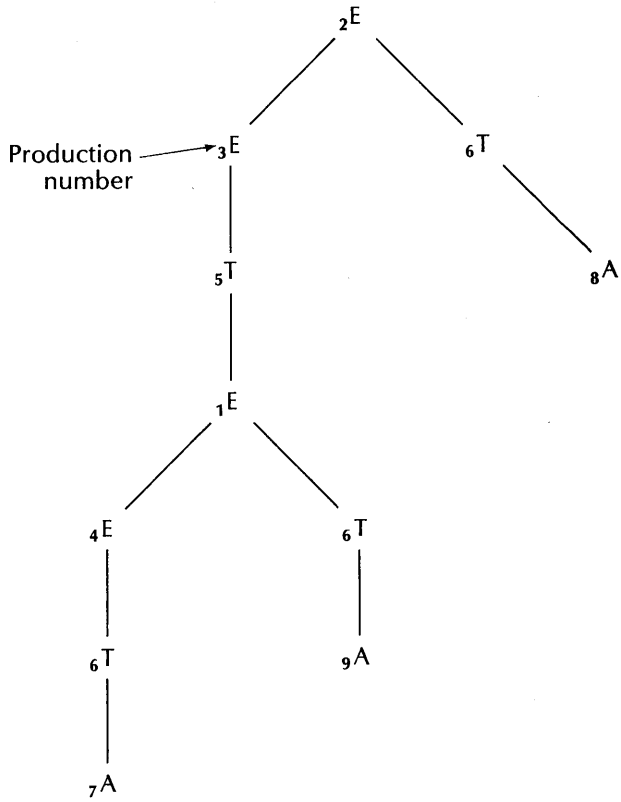


Figure 7.1(b). Derivation tree of figure 7.1(a) with terminal nodes removed.

$$E \rightarrow E + T, T\ E\ ADD$$

hence the children of this node become

$$T\ E\ ADD$$

T and E stand for a pair of subtrees that "move" with the permutation; this movement causes the string "z" to appear first in the translation although its corresponding source symbol "c" appears second in the source string.

The completed translation tree, figure 7.1(c) may now be scanned in left-to-right natural order to yield the translation string

$$z\ x\ ADD\ NEG\ y\ SUB$$

It is clear that the resulting translation string is independent of the order in which the translation is performed. Thus a bottom-up (right-most) or top-down (left-most) translation will yield the same translation string.

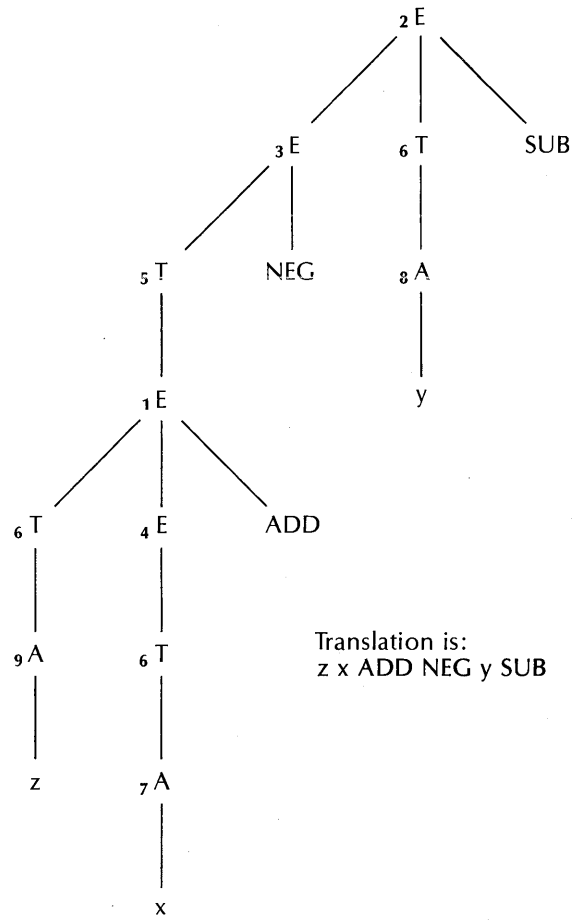


Figure 7.1(c). Derivation tree of figure 7.1(b) with nodes permuted and translation elements added, according to translation scheme.

Ambiguity

If the source grammar of a SDTS is ambiguous, then two or more derivation trees exist for some source string. Clearly, each of these derivation trees will yield a unique translation tree, however the two translations will in general be different.

We say that an SDTS is *ambiguous* if its source grammar is ambiguous and is *unambiguous* otherwise. It is clear that the addition of translation elements to a grammar cannot introduce or remove an ambiguity. However, it is possible that a translation can be unique despite a source grammar ambiguity; for example, if the translation element is identical to the source element, than every translation is unique despite an ambiguous source grammar.

Exercises

1. Develop a derivation and translation for the string

$$-c+[-b-a]$$

in SDTS S_1.

2. Prove that if the underlying source grammar G of an SDTS is unambiguous, then the translation string of any source string in L(G) is unique.

3. Given an SDTS S, how may an inverse SDTS S′ be defined? (If S translates a string "w" into "w'", then S′ translates "w'" into "w".)

7.2. Simple SDTS and Top-Down Transducers

A SDTS is said to be *simple* if the order of the nonterminals in the translation part of each rule is the same as the order of the nonterminals in the source part. For example, a rule

$$S \rightarrow S_1 + S_2, S_2\ S_1\ \text{ADD}$$

in a SDTS T causes T to be nonsimple.

A simple SDTS is such that no permutation of the derivation tree nodes is required for the translation. We have only removal of the source string terminals and insertion of the translation string terminals.

The significance of a simple SDTS for a top-down parser is expressed in the following theorem:

Theorem 7.1. If $T = (N, \Sigma, \Delta, R, S)$ is a simple SDTS whose underlying grammar is LL(k), then there exists a top-down deterministic push-down transducer that accepts any string in the input language of T and whose yield is the corresponding output string.

A *push-down transducer* (PDT) is analogous to the push-down automaton of chapter 4, except that it is permitted to emit a string of finite length (the translated output) upon each move. We define a push-down transducer P as follows:

A *configuration* of P is a 4-tuple (q, x, y, z), where q is a state in its finite control, x is the remaining input list yet to be scanned, y is a stack, and z is an output string emitted so far up to this point. Then in one move, we have

$$(q, ax, Yy', z) \vdash (r, x, gy', zz')$$

where there exists a machine rule

$$\delta(q, a, Y) \text{ contains } (r, g, z')$$

That is, given state q, with input symbol a, and stack top symbol Y, the machine is permitted to move to state r; in this move, the input symbol a is discarded, Y on the stack top is replaced by g, and the string z′ is emitted as output.

Then we say that w is an *output* for x if $(q_0, x, Z_0, \epsilon) \vdash * (q, \epsilon, u, w)$ for some state q and stack string u. State q_0 is the initial state and Z_0 is the initial stack contents; ϵ is the empty string. We say that P *halts by empty stack* if $u = \epsilon$ is a halting condition; alternatively, P *halts by final state* if q is a member of a defined set of halt states F, a subset of the machine's state set. These definitions are analogous to those of the PDA's defined in chapters 4 and 5.

We say that the PDT P is deterministic when both of the following conditions are met:

1. For all states q, strings *a*, and stacks Z, $\delta(q, a, Z)$ contains at most one element.
2. If $\delta(q, \epsilon, Z)$ is not null, then no symbol "a" exists such that $\delta(q, a, Z)$ is not null. That is, there must be no conflict between an empty and a nonempty move for some state and stack.

We may now describe the construction of a PDT P, given a SDTS T, where the underlying grammar of T is LL(1), such that P: (1) accepts every string in the underlying grammar of T, and (2) emits exactly the translation in T of every such string. The general case of LL(k), $k > 1$, is discussed in Lewis [1968].

Recall, from chapter 4, that the top-down LL(1) recognizer has two kinds of move:

1. An *apply* move, in which a nonterminal A on the stack top is replaced by a string "w", where $A \rightarrow w$ is some production rule. This operation is made deterministic through a selection table based on the stack top nonterminal A and the next input symbol.
2. A *matching* move, in which a terminal symbol "a" on the stack top is matched against the next input symbol. In a matching move, the stack top symbol is removed and the read head is advanced one symbol. A failure to match must be a syntax error.

Now a PDT operates as follows:

1. In an apply move, with a nonterminal A on the stack top, we know that a production $A \rightarrow w$ is involved, from the deterministic selector table. Let the translation rule in R be

$$A \rightarrow w, z$$

where

$$w = a_0 B_1 a_1 B_2 a_2 \ldots B_k a_k$$

and

$$z = b_0 B_1 b_1 B_2 b_2 \ldots B_k b_k$$

The B_i are nonterminals, the a_i are input alphabet strings (or empty), the b_i are output alphabet strings (or empty), and $k \geq 0$. Note that this translation rule is "simple."

We assume that the input and output alphabet symbols are distinguishable. Then A is replaced on the stack top by the composite string

$$b_0 a_0 B_1 b_1 a_1 B_2 \ldots B_k b_k a_k$$

with b_0 on the stack top.

2. If the symbol on the stack top is a member of the output alphabet, then it is removed and emitted as output.

3. If the symbol on the stack top is a member of the input alphabet, then it is matched against the input string (syntax error otherwise), it is removed from the stack, and the read head is advanced one position.

Example. Consider the simple SDTS

$$S \rightarrow 1\ S\ 2\ S\ ,\ x\ S\ y\ S\ z$$
$$S \rightarrow 0\ ,\ w$$

Note that we do not need to superscript the S's, since the SDTS is simple. The underlying grammar is obviously LL(1). We shall define the push-down tranducer and trace its operation for an example input string.

The transducer:

1. On stack top S, if the input symbol is
   (a) "1" then pop "S" and push "x1Sy2Sz"
   (b) "0" then pop "S" and push "w0"
2. On stack top in {x, y, z, w}, emit the stack top as output.
3. On stack top in {0, 1, 2}, match against the next input symbol.

A machine trace for sentence 1102020:

Operation	Remaining input	Output	Stack
Initially	1102020		S
1a	1102020		x1Sy2Sz
2	1102020	x	1Sy2Sz
3	102020		Sy2Sz

1a	102020		x1Sy2Szy2Sz
2	102020	x	1Sy2Szy2Sz
3	02020		Sy2Szy2Sz
1b	02020		w0y2Szy2Sz
2	02020	w	0y2Szy2Sz
3	2020		y2Szy2Sz
2	2020	y	2Szy2Sz
3	020		Szy2Sz
1b	020		w0zy2Sz
2	020	w	0zy2Sz
3	20		zy2Sz
2	20	zy	2Sz
3	0		Sz
1b	0		w0z
2	0	w	0z
3	ϵ		z
2	ϵ	z	rϵ

The output string is therefore "xxwywzywz". Note that the transducer accepts by empty stack.

If we did not have an LL(1) selector table, we would have a nondeterministic PDA. We have seen in chapter 4 that a PDA that accepts any context-free grammar can be constructed. We need only extend the notion of nondeterminism to a transducer, and this is easily done. Let *acceptance* of an input string mean that some set of choices based on stack top symbols eventually leads to an empty stack and input list. Then the emitted output is a translation.

If the underlying grammar is ambiguous, then no deterministic PDT exists, although a nondeterministic PDT exists; there may then exist more than one translation for one or more input strings.

If the underlying grammar is LL(1) then it is unambiguous, and there is exactly one translation for every acceptable input string.

The determinism of an LL(1) transducer means that the output can be immediately printed or punched on each move; it need not be saved on some erasable tape until the entire parse is complete. Output can also be generated as soon as a production is identified; the translator need not sustain output until the entire right part of the production is completely associated with the input list. Of course, it obtains this predictive ability through the one-symbol lookahead—the next symbol in the input list must be an infallible guide to the production rule about to come up, or else the grammar is not LL(1).

Among other things, determinism means that an entry in a symbol table could be made at the beginning, in the middle, or at the end of a production. Although we technically are discussing string-to-string translators, certain output string elements can in fact carry other kinds of operations, such as:

"enter the preceding identifier in a symbol table," or "turn off the compiler listing flag," etc. Such actions may be difficult or impossible to undo and must also be performed at the right time in the compilation. Such timing questions can be resolved for a LL(1) push-down transducer through a study of the production tree structure and the position of the action in some production.

Exercises

1. Trace the above transducer for the string 1021020.

2. Show that the PDT emits a correct translation for any acceptable input string.

3. Write a Pascal LL(1) translator. The program should include suitable data structure definitions.

7.3. Simple Postfix SDTS and Bottom-Up Transducers

A SDTS is said to be *simple postfix* if it is simple, and, in addition, every translation rule has the form

$$A \rightarrow a_0\, B_1\, a_1\, B_2\, a_2 \ldots B_k\, a_k\, ,\, B_1\, B_2 \ldots B_k\, w$$

That is, no translation terminals may appear in the translation element except as the right-most string "w". The significance of a simple postfix SDTS is expressed in the following theorem:

Theorem 7.2. For every simple postfix SDTS whose underlying grammar is LR(k), there exists a deterministic LR(k) push-down transducer that (1) accepts every sentence derivable in the underlying grammar and (2) emits as output the translation of that sentence.

Recall from chapter 6 that the LR(k) parser has three actions, selected by a current state and a next symbol, as follows:

1. *READ action:* the next input symbol must be among the terminal symbols associated with this state; if so, then advance the read head, push the next state S′ and go to state S′.
2. *APPLY action:* the stack top contains a set of states associated with the handle of some production i. The production number is specified by this state. Pop the states corresponding to the handle, push a state S′ determined from the exposed stack top state and the GOTO table, then go to state S′.

3. *ACCEPT action:* a goal-symbol production has just been reduced, and the stack contains one state corresponding to the goal symbol. Halt, with an indication that the input string is accepted. (We assume that the goal symbol never appears in the right-hand part of any production.)

These operations are directed by a finite-state control, with one or more states, that is permitted to examine as many as k symbols in the input in determining its action.

This machine may be easily transformed into a postfix transducer, by adding the following operation to the APPLY rule:

> Emit the terminal portion of the translation element associated with rule i.

We leave machine traces and a proof of theorem 7.2 to the exercises.

Theorem 7.2 has an important converse, as follows:

Theorem 7.3. There are unambiguous simple SDT's with an underlying LR(k) grammar that cannot be translated by any deterministic push-down automaton. (Note that this theorem has no LL(k) counterpart.)

This theorem emphasizes the importance of the postfix condition. For example, consider the SDTS T with the rules

$$S \rightarrow Sa,\ xS$$
$$S \rightarrow Sb,\ yS$$
$$S \rightarrow \epsilon,\ \epsilon$$

These rules are simple, but not postfix. It is easy to show that the underlying grammar is LR(1); however, no deterministic push-down transducer can be devised that will correctly translate its input language. Intuitively, it is necessary to emit an "x" or a "y" before the transducer can determine which of the three productions applies. Let us go through an example.

Consider the string "ba", which translates to "xy" (see figure 7.2.) A trace of the LR(1) machine apply states looks like this:

Stack	Input	List	Production
ϵ	ba		$S \rightarrow \epsilon$
Sb	a		$S \rightarrow Sb$
Sa	ϵ		$S \rightarrow Sa$

To emit the required string "xy", the machine must somehow predict that production $S \rightarrow Sa$ will eventually appear. What it in fact reports as apply states are the other two productions first.

1. $S \rightarrow Sa, xS$
2. $S \rightarrow Sb, yS$
3. $S \rightarrow \epsilon, \epsilon$

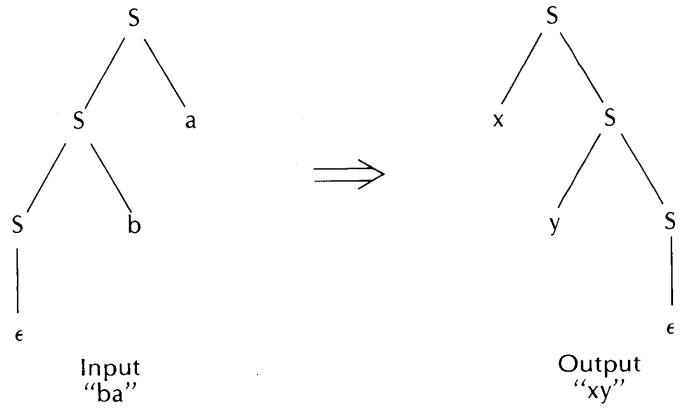


Figure 7.2. Translation of a string "ba" in a non-postfix simple syntax-directed translation scheme.

Now a top-down automaton could conceivably emit an "x" first, based on its start state and the first k input symbols. The top-down derivation of "ba" begins with $S \rightarrow Sa$; and "x" could therefore be emitted. However, this grammar is not LL(k) for any k; it contains left-recursive productions. This means that, for whatever k we choose, there is a string in the source language that will require a lookahead of at least $k+1$ in order to determine the next production in a top-down manner. We conclude that no deterministic transducer exists for this SDTS; the information that it needs in order to emit a translation can be arbitrarily far ahead in the input string.

It would appear that a simple postfix SDTS is more restrictive than a simple nonpostfix SDTS, and that a top-down SDTS is therefore somehow more powerful than a bottom-up SDTS. Such is not the case; this view overlooks the possibility of modifying the grammar in order to achieve the effect, if not the substance, of a simple nonpostfix SDTS.

Suppose that we want a translation output y within a translation element:

$$A \rightarrow x\ z, X\ y\ Z$$

where the rule $A \rightarrow xz, XZ$ is simple postfix and Z is nonempty. We may then consider the pair of productions

$$A \rightarrow x\ Y'\ z, X\ Y'\ Z$$
$$Y' \rightarrow \epsilon, y$$

Note that both productions are simple postfix. If the underlying source

grammar is still LR(k) with this change, then a bottom-up deterministic transducer will generate the desired translation. If the grammar is no longer LR(k), then it can be shown that the original grammar is not LL(k).

Another way to transform the productions is as follows:

$$A \rightarrow X'\ z,\ X'\ Z$$
$$X' \rightarrow x,\ X\ y$$

Here, we have assigned the string "x" to a new production, permitting the postfix translation Xy; the A production then becomes postfix as well.

The restriction to "simple" in both the bottom-up and top-down transducers is also easily removed by a system based on the SDTS transducers, as we shall see in section 7.4. Essentially, we may simply generate derivation trees (or better, abstract-syntax trees) and then rearrange the nodes internally as needed to achieve the necessary non-simple translation. Such a scheme may be limited in practice by the amount of random access storage space available for the trees; however, most modern computer systems are free of such limitations.

Examples. Consider grammar G_0 augmented with postfix translation elements given below:

$$E \rightarrow E\ +\ T\ ,\ E\ T\ ADD$$
$$E \rightarrow T\ ,\ T\text{—}$$
$$T \rightarrow T\ *\ F,\ T\ F\ MPY$$
$$T \rightarrow F,\ F$$
$$F \rightarrow A,\ A\ LOAD$$
$$F \rightarrow (\ E\ )\ ,\ E$$
$$A \rightarrow A\ a\ ,\ a$$
$$A \rightarrow a\ ,\ a$$

This SDTS is clearly simple postfix. A translation of the expression

$$aa * (aaa + a)$$

is easily shown to be

$$aa\ LOAD\ aaa\ LOAD\ a\ LOAD\ ADD\ MPY$$

and is exactly that required by a postfix stack machine. The LOAD operation operates upon the preceding identifier. If we wish LOAD to operate on the following identifier instead, we may modify one of the F productions as follows:

Instead of production $F \rightarrow A$, A LOAD, we introduce the two productions

$$F \rightarrow L\ A,\ L\ A$$
$$L \rightarrow \epsilon,\ LOAD$$

The LOAD will clearly then precede its identifier. The grammar is still LR(1); the reduce operation for the empty production $L \rightarrow \epsilon$ can be inferred from a one-symbol lookahead of the following identifier.

The translation of aa*(aaa+a) will then be

LOAD aa LOAD aaa LOAD a ADD MPY

As another example, consider the common control statement

$S \rightarrow$ if E then S else S

where E is an expression and S is a statement. In this form, a simple postfix translator can only generate evaluation code for E, then evaluate the two statements S. Only when both of these are complete is it known that an if-then-else control statement is being parsed. But such a statement requires the emission of a conditional branch instruction between the E and the first S, and an unconditional branch instruction between the two S's. How might this be done?

One solution is to partition this production into three productions:

$S \rightarrow$ I else S
$I \rightarrow$ T then S
$T \rightarrow$ if E

Clearly, the three put together are equivalent to the original production. However, by so separating the three productions, we may generate a simple postfix translation as follows:

$S \rightarrow$ I else S,	I S ; L2 :
$I \rightarrow$ T then S,	T S ; BR L2; L1 :
$T \rightarrow$ if E,	E ; BRF L1

Here, "BR L2" means "branch to the statement labeld L2." "BRF L1" means "branch to L1 if the stack-top expression is FALSE, otherwise continue. Delete the stacktop in any case."

The semicolons will appear as separators in the translation strings.

Then the statement

if a then S1 else S2

would translate to

```
LOAD a;
BRF L1;
S1;
BR L2;
```

```
L1:  S2;
L2:
```

Here, L2 marks whatever statement follows the if-then-else.

Given the same if-then-else structure, we may achieve the same output by introducing productions for the "then" and "else" keywords, e.g.

S → if E H S L S ,	E H S L S ; L2 :
H → then ,	; BR L2 ; L1 :
L → else ,	; L2 :

The effect of the added productions is clearly to bring out some translation strings at the desired point in the parsing of the whole statement.

Finally, empty productions may be used:

S → if E then H S else L S ,	E H S L S ; L2 :
H → ϵ ,	; BR L2 ; L1 :
L → ϵ ,	; L2 :

These three translation schemes are equivalent and are LR(1), assuming that the remainder of the grammar is reasonable.

As an example of one limitation of a simple postfix SDTS, consider a function call, with the underlying source grammar:

```
F → A ( L )
L → L ; E
L → E
A → A a
A → a
```

The call parameter list L consists of parameters E separated by semicolons. The usual separator is a comma, but we are using a comma as a metasymbol.

For discussion purposes, we would like the procedure call to appear at the end of the code for the procedure parameters; this is how procedures are called on many (but not all) machines. However, the procedure name appears first in the syntax. We therefore need a nonsimple transducer if the call and the call name is to appear after the actual parameter coding.

However, a postfix call may be acceptable on certain machines, and is easily generated by the simple postfix grammar:

$$F \rightarrow A\ (\ L\ )\ ,\ A\ L\ ;\ CALL$$
$$L \rightarrow L\ ;\ E\ ,\ L\ E$$
$$L \rightarrow E\ ,\ E$$
$$A \rightarrow A\ a\ ,\ A\ a$$
$$A \rightarrow a\ ,\ a$$

Then the function call

aa(a, aaa + aaaa)

will translate to

aa; LOAD a; LOAD aaa; LOAD aaaa; ADD; CALL

and the CALL operates on the very first identifier, aa. Nothing can be done using a simple SDTS to emit the procedure name "aa" after its CALL.

7.4. A General Transducer

A simple SDTS can be implemented by a push-down stack automaton, augmented with a translation string emitter, as we have shown in the preceding two sections. A non-simple SDTS is easily implemented by constructing a rearranged tree for the translation string as part of the parsing stack operations. When the parse is complete, the translation tree may be scanned in natural order to yield the translation string. Of course, the resulting machine is no longer a PDA.

The system presented in the following algorithm assumes that the base grammar is LR(k). However, the extension to any base grammar or to other parsers is trivial.

Let M be a translation machine that carries two stacks, one the usual LR(k) viable prefix stack (the *parse* stack) and the other a stack of translation subtrees (the *translation* stack). During the parsing process, each translation subtree will correspond to a nonterminal in the parse stack, and will represent a translation tree valid for that nonterminal and the derivations that previously stemmed from it. The translation stack will carry the roots of translation trees. Each nonterminal in the parse stack corresponds to a root in the translation stack.

Machine M has the actions READ, APPLY, and ACCEPT as follows. (The ERROR action in an LR parser is of no special interest just now.)

1. *READ:* The next input symbol is shifted into the parse stack. The translation stack is unaffected. A terminal in the parse stack does not have a corresponding translation stack element.

2. *APPLY:* Suppose we have the translation rule

$$N \rightarrow a_0\ A_1\ a_1\ A_2 \ldots A_m\ a_m,\ w_0\ B_1\ w_1 B_2 \ldots B_m\ w_m$$

Remove the string "$a_0\ A_1 \ldots A_m\ a_m$" from the parse stack, but not the associated subtrees (yet). The translation stack is associated with trees rooted in the A_i. Remove these trees from the translation stack and permute them according to the translation rule; they now correspond to the B_i. Create a new tree root N′ and new tree leaves for the translation elements w_i and arrange these two sets of trees as the children of the new node N′. Now push N′ into the translation stack.

At the end of the APPLY step we have a new subtree rooted in N′. The children of N′ consist of leaves and subtrees, corresponding to the elements of the translation part of the rule, i.e., to $w_0\ B_1\ w_1 \ldots B_m\ w_m$. The B_i have been obtained from the translation stack and the w_i have been newly created.

3. *ACCEPT:* The tree associated with start symbol S in the parse stack is the translation tree. It may be scanned in left-to-right natural order to yield the translation string.

This translator is illustrated in figure 7.3 for the nonsimple SDTS

1. S → aSA , 0AS
2. S → b , 1
3. A → bAS , 1SA
4. A → a , 0

and the input string "abbab". The underlying grammar is SLR(1). Only the apply steps are shown. For example, just before the first apply action, the stack contains "ab" and the input is "bab". Rule 2, when applied, causes "b" on the stack top to be replaced by "S", linked to a son "1". String "ba" then is shifted into the stack in preparation for the next apply action.

Just before the last apply step, the parse stack contains aSA and the translation rule is

$$S \rightarrow aSA\ ,\ 0AS$$

A new leaf is created for the 0 of the translation element, and the A and S trees are interchanged. The stack top trees are replaced by a tree representing the translation subtree corresponding S, in this case, the complete translation tree. This tree may clearly be scanned from left to right to yield the translation string "01101".

A practical tree structure suited to this algorithm is illustrated in figures 7.4 and 7.5. Each translation stack element is a pointer to a *tree stack* that in turn carries a *mark*, a *token* and a *child* within each element. The children of some node are arranged contiguously in the tree stack, and only the right-most child carries a mark, indicating that it is the last child. If a tree node N is the root of a subtree, then N carries a child pointer, otherwise its child pointer cell is empty.

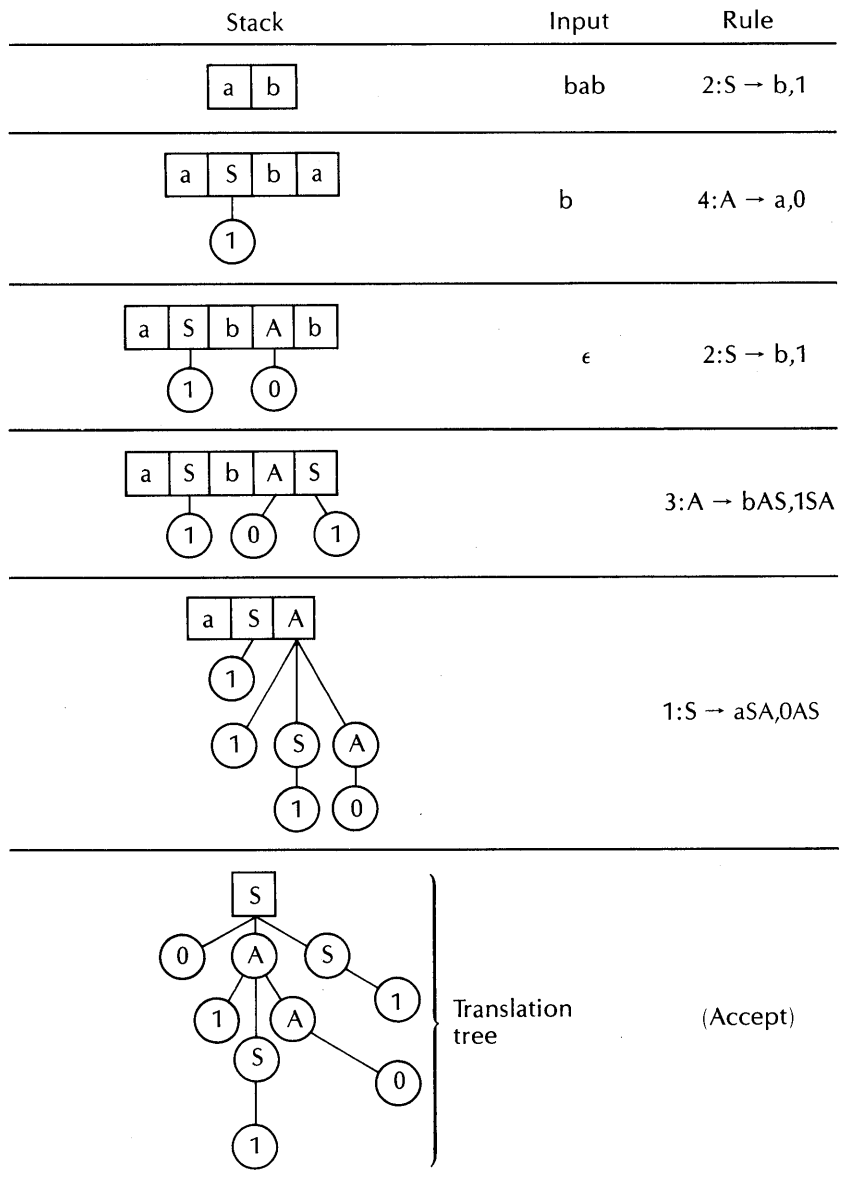


Figure 7.3. Translation by tree construction. Only the apply steps are shown.

Thus, in figure 7.4, the tree rooted in S has the children "0AS", in tree stack nodes 6, 7, and 8. Node 7 is the root of a subtree in which level 1 is "1SA". Figure 7.5 shows the tree structure represented by figure 7.4.

An empty leaf node must be specially denoted by a null token, since a leaf

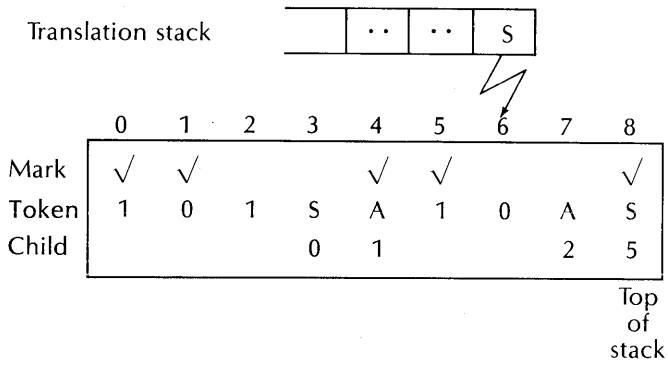


Figure 7.4. A compact representation of the tree of figure 7.4.

must occupy at least one cell to carry a mark.

This representation is easily generated during translation, since all the children of some node can be allocated at once in the same APPLY operation and may therefore be contiguously arranged without any movement of nodes in the stack.

We may therefore modify the tree building algorithm as follows:

Introduce a third stack of three elements each: a mark, a token, and a child pointer; call it the *tree* stack. We really don't need the token, but we shall eventually want to use it to associate more information with the translation tree. Then modify the APPLY step as follows:

2. *APPLY:* Given the translation rule

$$N \rightarrow a_0\, A_1\, a_1\, A_2 \ldots A_m\, a_m,\ w_0\, B_1\, w_1 B_2 \ldots B_m\, w_m$$

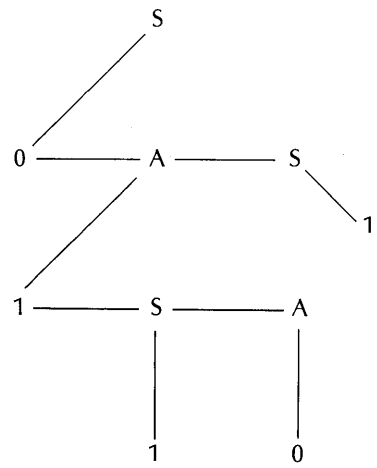


Figure 7.5. Schematic form of the tree of figure 7.4.

the translation stack will contain pointers to subtrees corresponding to $A_1, A_2, \ldots, A_m$. Augment the tree stack with sufficient elements to contain all the symbols in the translation part $w_0\ B_1 \ldots B_m\ w_m$. Mark the top of the tree stack. Set the pointers associated with the w_i to null. Set the pointer associated with a node B_i to the pointer in the translation stack corresponding to A_j, where A_j is associated with B_i in the translation rule. Finally, remove the top m elements from the translation stack, and replace it with a pointer to the left-most new element in the tree stack.

Exercises

1. Devise a simple postfix SDTS for each of the following structures. V is a variable, E is an expression, S is a statement, and L is a statement label.

   S ::= while E do S
   S ::= for V:=E until E do S
   S ::= L: S
   S ::= case E SL endcase
   SL ::= SL ; S | S {statement list}

2. Write a Pascal program that implements the apply step of the general transducer given in the previous section.

3. Devise data structures to facilitate a table-driven PDT that generates a translation tree.

4. State an SDTS implementation for an LL(1) source grammar, based on a tree builder.

7.5. String Transducers and Their Limitations

We have seen how an SDTS transducer can be applied as a string-to-string translator. Such a translator has certain practical applications, especially if extended by incorporating symbol table functions and some simple arithmetic. Some compilers are essentially string translators, from a high-level source language to symbolic assembly language for a machine. The advantage of writing such a compiler is its very low cost. The translator can be written as an SDTS and interpreted by a simple general purpose system.

We have also seen that an SDTS transducer can be used to construct a translation tree directly (section 7.4). We may modify that algorithm and generate an abbreviated tree, called an *abstract-syntax tree*; this notion will be expanded upon in section 7.5.2.

Finally, rather than simply emit a string, certain string tokens or all the tokens can be interpreted as synthesis actions.

7.5.1. String Translators

Let us assume that a general SDTS transducer is available, such as that described in section 7.4. What sorts of translations can be achieved and what are its limitations?

A great deal of literature exists on translator writing systems, or compiler-compilers, e.g., Feldman [1968] or, more recently, Griffiths [1974c]. A translator writing system, or TWS for short, often contains a string translator as its core, but is augmented by a variety of other needed functions.

Roughly speaking, a string translator can accept any source language defined by a context-free grammar, preferably LR(1), and emit a translation defined by the SDTS structure. The translation language may bear scant resemblance to the source language. Metcalfe [1964], for example, illustrates a translator of some simple English sentences to German, an operation that requires some rearrangement of sentence structure.

Unfortunately, the problem of language translation is not so easily solved. Natural languages are not context-free, nor are the common computer artificial languages. Both require some auxiliary mechanisms, in the form of a symbol table or a dictionary, at least, to achieve a reasonable translation.

Let us examine some of the translation needs that cannot be satisfied by a pure string translator.

Data Types

Data variables are usually typed, e.g.,

```
var I: integer; var R: real;
```

The effect of such statements is to associate one or more attributes with the identifiers I and R, as well as allocate space for them. Now a variable reference, such as:

```
R:=I+1;
```

usually requires knowledge of the variable types to generate correct code. Thus knowing that I is type "integer" and R is type "real," a compiler might generate the assembly code

```
LOAD   I    {fetch variable I to top of stack}
LOAD   =1   {push a 1 into stack}
ADD         {e.g., integer add}
FLT         {convert to real}
STD    R    {store double value on stack top in R; the "float"
            converts from a single to a double value}
```

for a stack machine. The choice of each instruction in this sequence depends on the types of the two variables.

Typing is achieved through a token table mechanism, and without some extensions to a string translator, we cannot hope to generate suitable translation sequences that consider variable typing.

However, typing can be bypassed in one of these ways:

1. The language may have only one type, for example, "real," so that every operation is determined only by the algebra and not the variables.
2. The machine representation of a data item may contain a data type descriptor along with the data item. Then all the necessary conversions and choice of the appropriate operation becomes a machine function and not a compiler function. For example, a single generic "add" instruction might be interpreted by the machine as a "real" add, an "integer" add, etc., depending on the operand descriptor.
3. The translation may be to a high level language that supports the features of the simpler language. The syntax may be different, but if the object language contains all the features and semantics of the source language, then a straightforward string translation is feasible.

However, variable typing occurs in most programming languages, and must usually be dealt with.

Array Dimensions

Consider an array declared as follows:

```
array X[0:25, 0:35] : real;
```

A reference to this array of the form X[I, J] may require knowledge of one or the other dimensions, depending on the storage conventions for multidimensioned arrays in the target machine system. For example, the array may be stored in rows in one long linear array. An offset N of the form

```
N:=36*J+I;
```

must then be computed for the reference X[I, J]; the "36" is the second dimension plus 1 (because of zero basing). Again, a simple string translator cannot supply the "36" without some help.

Nevertheless, there are ways:

1. Given an array declaration, a string translator can create a constant declaration, by making up new names based on the old ones, e.g.

```
source:  array X[0:25, 0:36] of real;

object:  array X[(25-0+1)*(36-0+1)] of real;
         const #X0=35-0+1;
```

Then the new name #X0 can be used in a variable reference of the form X(J, I) in place of the needed dimensional factor 36:

```
N:= #X0*J+I;
```

However, this approach to array indexing supposes that the object language translator can support a class of manufactured names and constant expression structures, which is seldom the case.

2. An array might be supported by a *dope vector* at run-time that is associated with the array elements and that contains the array dimensions. Then an array reference can be handled by a call to some general purpose procedure that accepts the indices I and J and a reference to the dope vector.

3. A multidimensioned array could be accessed through a transfer vector system. Consider a reference to X[I, J] again, and let there be a vector of 36 pointers (corresponding to the 36 possible J values). Each of the 36 pointers points to an I vector, e.g., one slice of the array. Then a fetch of X[I, J] could be coded without knowing the J dimension: indirect through V(J), then indexed by I. For some machine architectures, this kind of array access may be considerably more efficient than one based on a calculation of a linear array index.

Despite these solutions to array indexing, there is really no practical alternative to a symbol table system.

Branches and Procedure Calls

A string translator can only organize branches and procedure calls symbolically; without arithmetic or a token table, it cannot fix any program addresses. Furthermore, a string translator cannot ascertain whether a given branch label or procedure name has appeared more than once, so that an error of this kind cannot be reported until the object translation occurs.

Parameter Passing

The parameters of a procedure are usually of several different kinds (reference, value, name, etc.). Thus the generated code for the actual parameters depends on the declarations of the formal parameters. This dependency is another area of deficiency of a simple string translator—it must somehow generate object strings for actual parameters that are acceptable regardless of the declarations of the corresponding formal parameters, which may be impossible to do.

Some languages, such as Fortran, have only one passing mechanism —reference. In that case, a string translator can generate acceptable code.

In any case, an error in the number of actual parameters cannot be detected until the object is translated.

If a function call and an array reference appear syntactically the same, the object language of a string translator must be indifferent to the two. Thus in the Fortran statement

$$Y = X(I, J)$$

the X(I, J) could be an array reference or a function call. The two cases are distinguished through a prior declaration of X as an array or a function. A symbol table is necessary to associate such an attribute with each identifier.

7.5.2. Abstract-Syntax Tree Construction

An *abstract-syntax tree* or AST is a condensed tree representation of some language structure. It contains only that information needed for the remaining transformations or reductions of the structure. Any language structure can potentially be represented as an AST, e.g., expressions, control statements, input or output statements, and declarations.

An AST can serve as an intermediate structure in a partitionable compiler. The compiler system that generates it might be one distinct piece of software, and the system that interprets it and generates code from it may be another distinct piece. By so partitioning the compiler software for different languages, a given code generation piece might be used with different AST generation pieces. Of course, the code generation piece must accept any AST that any of the AST generators can construct, which will increase its complexity.

We shall see that code generation and certain optimizations are facilitated by an AST representation.

Consider grammar G_0:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow a$

The derivation tree for even simple expressions is rather large. For example, the expression "a*(a+a)" has the derivation tree shown in figure 7.6. This tree has 7 leaf nodes and 11 interior nodes, yet the expression itself contains only two operators and three operands. Why all the complexity?

Most of the complexity arises from the form of the grammar. Every derivation step is reflected in a tree node, yet many of the derivation steps are merely to provide a suitable precedence for the operators + and * and to cause operations inside parentheses to be performed before those outside.

We can reduce the derivation tree to an AST by first removing all the tree links associated with the single productions, i.e., $E \rightarrow T$, $T \rightarrow F$, and $F \rightarrow a$. They serve a purpose in the derivation process, but no useful role in the final structure.

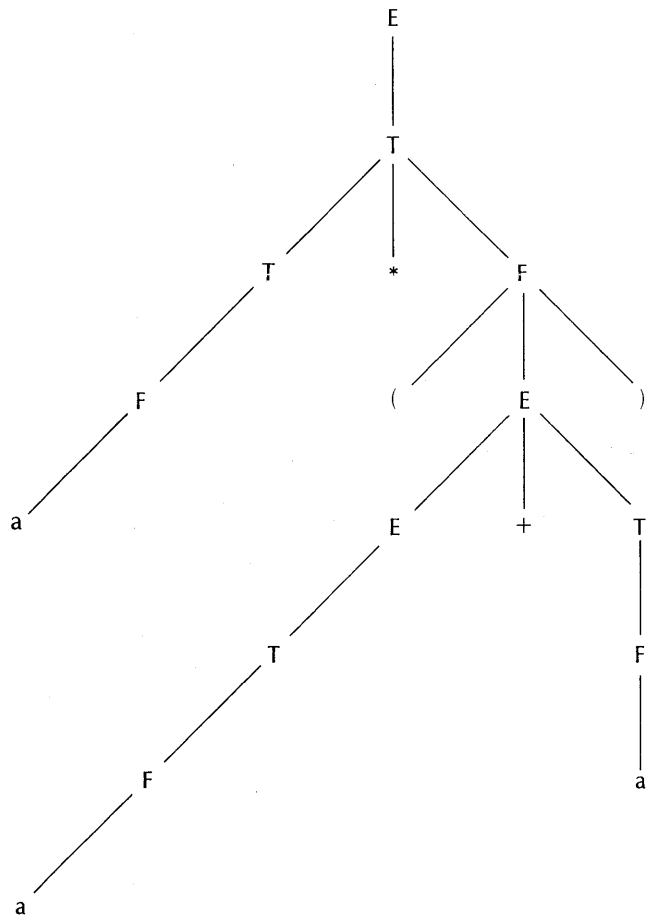


Figure 7.6. Derivation tree in G_0 for a simple expression a*(a+a).

Next, the tree nodes containing the operators * and + can be reduced to a single node containing only the operator. We clearly no longer have a derivation tree, but so far have not lost any structural information either. At this point, our reduced tree looks like figure 7.7. It still says that the addition must be done first, then the multiplication. What about the parentheses? If we collapse the parenthesis production F → (E) will the operator ordering established by the parentheses be lost? The answer is no—the "a" node inside the parentheses came from a sequence of derivation steps that have already provided for the precedence of the + operator inside the parentheses over the * outside. We arrive at the AST in figure 7.8.

This tree is also the abstract-syntax tree for the expressions

a*((a)+a)
a*(a+(a))
(a*(a+a))

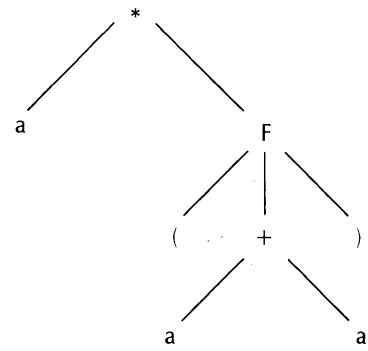


Figure 7.7. Derivation tree of figure 7.6 reduced by collapsing single productions and operator productions.

as should be obvious. Each of these is mathematically equivalent; we conclude that an abstract-syntax tree for an expression is a kind of canonical form for an equivalence class of expressions. However, the AST equivalence class is not complete, for it does not take into account commutativity or associativity of the operators + and *. Consequently, the following expressions, although mathematically equivalent, yield different AST's:

a*b + c,
c + b*a,
c + a*b, etc.

We can easily generate an AST directly by some simple changes in the SDTS implementation. Instead of generating a complete translation tree, we simply generate an AST by pointer mechanisms similar to those in section 7.4. A grammar and the corresponding tree translation elements are illustrated in figure 7.9.

For a bottom-up parser, the rules in figure 7.9 can be understood as follows. Consider the E → E+T rule. When E+T appears as the handle on the top of the parser stack, there will be two subtrees connected to the E and the T elements. (Nothing is connected to the "+".) The translation rule

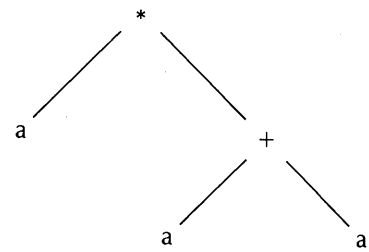


Figure 7.8. Abstract-syntax tree for expression a*(a+a).

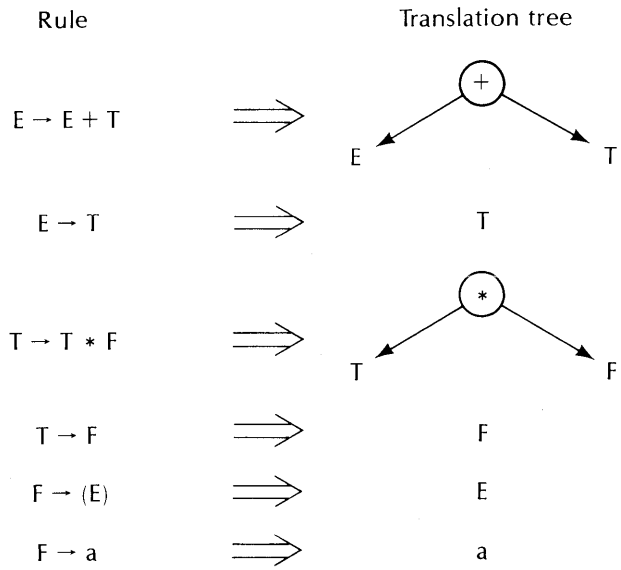


Figure 7.9. Direct generation of an abstract-syntax tree from grammar G_0 and its parser.

effectively says: "Create a new node, labeled '+', and give it two sons—the E and T subtrees. We now have a new tree rooted in the '+' node; attach it to the E that replaces E and T on the stack."

The translation rule E → T effectively is applied as follows. When T is the handle, it is attached to some subtree; merely attach this subtree to the E that replaces T in the stack.

Finally, the translation rule F → (E) calls for attaching the E subtree to the element F on the stack, and the rule F → a calls for attaching the terminal token "a" as a tree consisting of one node to the element F in the stack.

When the "accept" state is reached, we will have the AST attached to the start token on the stack top.

The general form of an AST is a tree whose interior nodes are operators and whose leaves are simple variables or constants. The operators may be of any variety, unary, binary, or n-ary. For example, an array reference can be represented by the AST shown in figure 7.10, where X is a multidimensional variable, and $e_1, e_2, \ldots, e_n$ represent expression AST's for the indices; the reference in source form would look like this:

$$X(e_1, e_2, \ldots, e_n)$$

A procedure call AST would be identical in form, except that the operator would be different, of course, and a different class of actual parameters might be permitted by the language.

An AST can represent control structures and declarations as well. For example, a CASE statement can be modeled as an AST as shown in figure

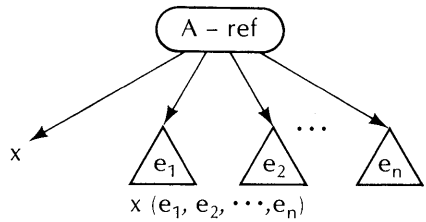
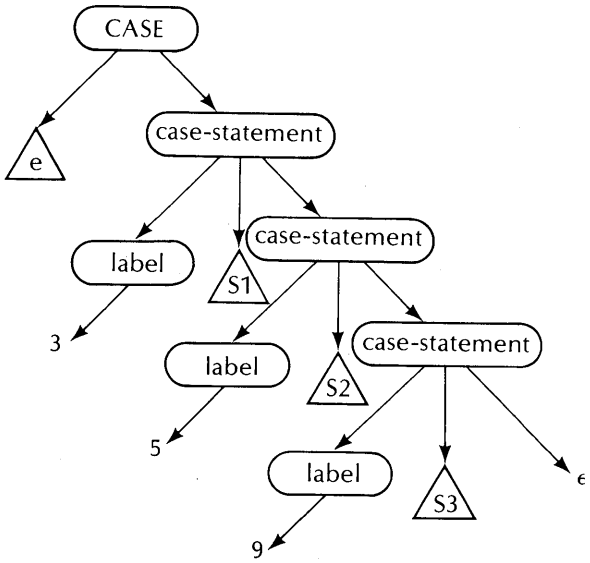


Figure 7.10. An array reference or procedure call abstract-syntax tree.

7.11. The nodes labeled "case statement" can all be the children of the CASE node, or can form a parent-child chain as shown, the last one with a nil right-most child.

Considerable freedom in the definition of an AST exists. However, an AST should satisfy several properties:

1. Each interior node must carry an operator with uniquely defined properties, e.g., number, kind, and ordering of children.




```
CASE e OF
  3: S1;
  5: S2;
  9: S3
END;
```

Figure 7.11. A CASE statement expressed as an abstract-syntax tree.

2. Every operator should carry some semantics and not just be a "place holder." The test is whether it can be removed without losing information in some way.

3. An operator usually arises through some production with either an empty right member or more than one right member token. A single production almost invariably contributes nothing to the AST construction. This rule is not absolute, but is a reasonable guideline in designing an AST and linking it into a translation scheme.

Exercises

1. Write a set of productions for a case statement such that the component statements must carry one label each. Then devise an AST generator for your set.

2. Devise suitable data structures through which a table-driven, AST-generating, bottom-up SDTS can be designed. State the table-interpreting algorithm.

7.5.3. A Practical Bottom-Up Synthesis System

Figure 7.12 shows a set of three push-down stacks that can serve as the basis of a general purpose, bottom-up synthesis system. Each of the stacks grows downward, i.e., the "top-of-stack" is the bottom element.

Stack STATE is the usual LR(k) push-down stack. It contains state

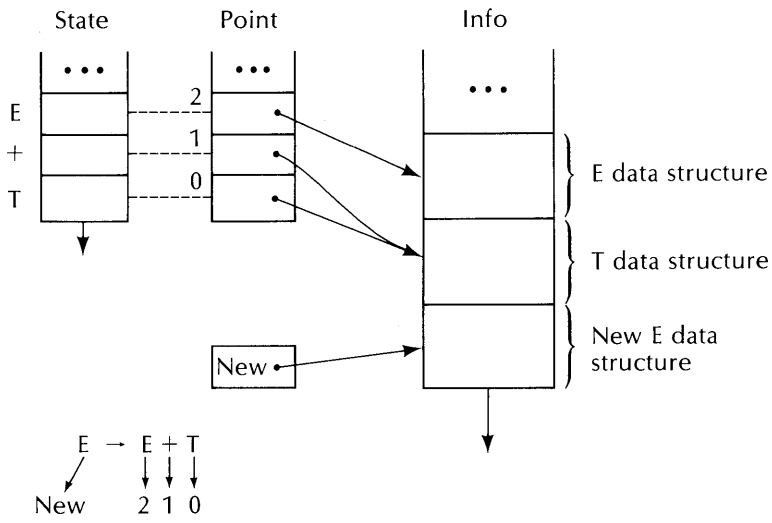


Figure 7.12. A generalized bottom-up translation system with stacks.

numbers used in the parsing process. On an APPLY step, the topmost set of state numbers corresponds to the handle. We show an APPLY of production $E \rightarrow E+T$ in figure 7.12.

Stack POINT contains pointers to an information stack INFO. Each element in POINT corresponds to an element in STATE. Most of the terminal elements and some of the nonterminal elements have an empty data structure in INFO, for example, the terminal token "+", which points to the same element as the nonterminal "T". Since some terminal elements carry information (e.g., identifiers), the POINT stack will correspond exactly to the STATE stack.

In addition to the three stacks, one cell for a pointer NEW is provided.

Figure 7.12 shows the state of the system just before an APPLY operation is executed. Symbols E and T correspond to two data structures in INFO, pointed to by POINT[2] and POINT[0], respectively. An empty area for a "new" E (corresponding to the left part of the rule $E \rightarrow E+T$) has been provided on the INFO stack top, and pointer NEW is assigned to it.

We may now perform some sort of operation on the old E and T data structures, either modifying them, or using them to construct a new data structure. There are three general end results of such operations, illustrated in the next three figures.

Figure 7.13 shows a PASS. The new data area and all but the deepest in the stack of the old areas are dropped. Two elements are dropped from the POINT and STATE stack in the example. Note that POINT[0] continues to point to the same structure as before. This operation is useful for single productions of the form $A \rightarrow B$, requiring no action. It can be seen that the data structure associated with B is simply "passed along" to A. Of course, the data carried by the E structure may be modified.

Figure 7.14 shows a KEEP. Here, we keep the old data structures and the new one (presumably filled with useful information by this time). However,

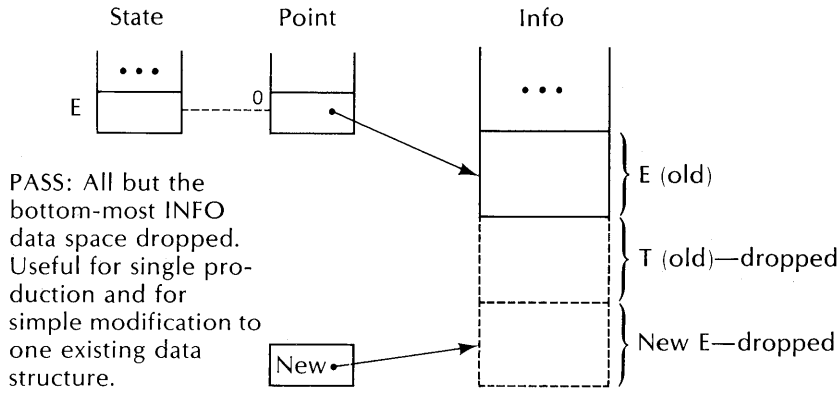


Figure 7.13. The PASS operation (see figure 7.12).

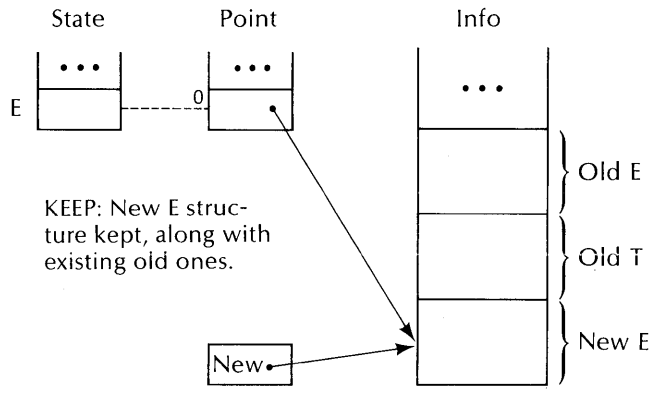
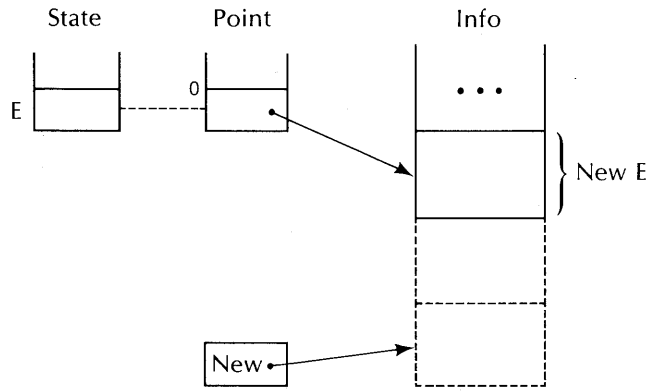


Figure 7.14. The KEEP operation (see figure 7.12).

the old pointers in POINT are lost and replaced by the NEW pointer to the new one. This operation is useful in building a tree structure on the stack. The new E structure could carry an operator (e.g., "+" in this example) and pointers to the old E and T structures. Note that we have an opportunity to rearrange the old E and T children of the new parent node E.

Figure 7.15 shows a REPLACE. Here, information in the old data structures is used to construct a new one, but then the old structures are discarded by "sliding" the new structure into their place. The old pointers in POINT are discarded and replaced by a pointer to the new structure. The net effect is the same as a KEEP, except that useless old information is removed from INFO, releasing space for new information.

The use of REPLACE implies that the replacement information must be



REPLACE: New E area kept, but old areas removed and stack reduced.

Figure 7.15. The REPLACE operation (see figure 7.12).

built in some temporary space while the existing space is being accessed; then the temporary space is moved into INFO.

Fortunately, REPLACE is unnecessary in many languages. Suppose that the grammar contains a nonterminal N such that N's data structure is empty, and the data structure of every nonterminal X that can precede N in the stack is also empty. Then the INFO stack may simply be reset on any production N→w, and no special attempt to reclaim space need ever be made. For example, in Fortran or Basic, the INFO stack may be cleared at the end of every logical line (a line with its continuations). In Algol, the INFO stack might be cleared at the end of every statement or declaration.

Of course, the STATE stack cannot be reset in Algol; it will contain essential information regarding the blocks.

Organization of a Synthesis System

We may now describe the organization of a synthesis system for a compiler as follows:

1. Design a data structure for each of the grammar's tokens, terminal and nonterminal. (Many of these will be empty.) Each structure will occupy a certain number of words or bytes in INFO. Some may occupy a variable number, however, the system can be more efficient if each structure has a fixed length.

   For example, if the scanner recognizes each identifier as a unit (i.e., the grammar contains <identifier> as a terminal token), then the data structure for <identifier> might be:

   ```
   type IDENT=record LENGTH: integer;
                     NAME: array[1..15] of char
   end
   ```

   As another example, suppose the grammar contained a production

   <array-spec> ::= array [<number> : <number>] of . . .

   (to simplify a Pascal structure), where the two <number> elements are also nonterminals returned by the scanner as numbers. Then a suitable data structure for <array-spec> might be:

   ```
   type ARRAYSPEC = record LOWER: integer;
                           UPPER: integer
   end
   ```

2. Upon an APPLY, the pointers in POINT will correspond to certain data structures in the INFO stack. Pascal provides no direct means of

associating a POINT pointer with a type, but we may get around that by statements of the following kind:

```
var S0: ↑EXPR;        {new pointer to an EXPR type}
S0:=POINT[0];         {pointer now given a value}
```

(The second statement requires a Pascal compiler extension.) Similarly, pointer NEW must be associated with the data structure of the left part of the production rule. We could devise a system that performs such pointer assignments automatically, or simply do it as needed as part of each apply action.

3. We next design a synthesis operation for each production. The nature of the operation is usually clear from the data structures involved, and what we wish to accomplish at the moment at which that particular production is involved in the bottom-up parse.

   For example, each of the productions $E \rightarrow E+T$, $T \rightarrow T*F$ will call for adding a node to an AST being built in INFO. A production of the form

   $$<\text{statement}> ::= <\text{identifier}> := <\text{expr}>$$

   will call for evaluation (through code emission or interpretation) of the <expr> AST, and then emitting a "STOR <identifier>" instruction.

4. At the end of the synthesis operation, one of the three operations KEEP, REPLACE, or PASS is selected.

The parser will manage the STATE and POINT stacks. A parser SHIFT step may require the introduction of a terminal token data structure into INFO; this structure must therefore be supplied by the scanner and written into INFO.

Example. Let us build an AST with this system for the grammar G_1 given below:

```
1. E → E + T
2. E → E - T
3. E → - T        {unary minus}
4. E → T
5. T → T * F
6. T → T / F
7. T → F
8. F → ( E )
9. F → <ident>
```

10. F → <ident> (E) {an indexed reference}
11. F → <const> {a decimal number}

Grammar G_1 is clearly an extension of G_0 to include subtraction, division, unary minus, identifiers, and indexed identifier references.

We need only one data structure, an EXPR type with several cases as follows:

```
type EXPR = record case CODE: set of (binop, unop, var,
                                      inxvar, const) of
      binop: (OPERATOR: set of (add, sub, mlt, div);
              LEFT: ↑EXPR;
              RIGHT: ↑EXPR);
      unop: (CHILD: ↑EXPR);    {only one unary operator: −}
      var: (SYMPNTR: ↑SYMTAB); {pointer to token table}
      inxvar: (SYMPNTR: ↑SYMTAB;
               INDEX: ↑EXPR);
      const: (CONSTVAL: integer)
end
```

The apply operations for each production may now be written as follows. We have written out specific assignments to the pointers S1, S2, etc., corresponding to the POINT stack contents. This could have been done automatically before each apply operation.

Production 1: E → E + T

```
with NEW do
begin {NEW assumed set to top of INFO stack}
  CODE:=binop;
  OPERATOR:=add;
  LEFT:=POINT[2];
  RIGHT:=POINT[0];   {pointers to the left and right
                      expressions}
end;
KEEP; {this builds upon existing structures}
```

Production 2: E → E − T

Same as production 1, except "sub" instead of "add."

Production 3: E → − T

```
with NEW do
begin
  CODE:=unop;
  CHILD:=POINT[0];    {pointer to T part}
end;
KEEP;
```

Production 4: E → T

```
PASS;    {amounts to a do-nothing}
```

Production 5: T → T * F

Same as production 1, except "mlt" instead of "add."

Production 6: T → T / F

Same as production 1, except "div" instead of "add."

Production 7: T → F

```
PASS;
```

Production 8: T → (E)

```
with NEW do
begin
   POINT[2]:=POINT[1];
```

Production 9: F → <ident>

```
with NEW do
begin
   CODE:=var;
   SYMPNTR:=FETCH(POINT[0]);
REPLACE;  {essentially a PASS, except that we want a
                pointer to the E, not to the left parenthesis}
end;
```

```
                {FETCH is a token table function. It locates the
                 identifier pointed to by POINT[0] and returns a
                 pointer to its attributes. Hence we have effectively
                 attached the attributes of the identifier to the
                 data structure being built. Failure to find the
                 identifier is an error condition that we shall not
                 deal with here.}
end;
REPLACE;    {we don't need the identifier name anymore}
```

Production 10: F → <ident> (E)

```
with NEW do
begin
   CODE:=inxvar;
   SYMPNTR := FETCH(POINT[3]);  {the <ident> has index 3}
   INDEX := POINT[1];
end;
KEEP;     {keep the new one and previous ones}
```

Production 11: F → <const>

```
type  C=CONSTANT: integer;
var P : ↑C;    {pointer to type C}
with NEW do
begin
   CODE:=const;
   P:=POINT[0];
   CONSTVAL:=P↑.CONSTANT;  {fetch the constant value
                            from INFO}  end;
REPLACE;  {don't need the old constant anymore}
```

Figure 7.16 shows the structure of INFO upon parsing the expression I1(X−15)+IM2. POINT[0] points to a binary ADD node, with left child I1(X−15) and right child IM2. The IM2 node is a "variable" node, pointing to the attributes of IM2 in the token table. The left child node I1(X−15) is an "inxvar" node, etc. We clearly have an AST, and the identifier attribute association has been made.

Evaluation of an AST

Now suppose that the AST is evaluated by some tree-walking or transforming automaton. The evaluation yields a block of output code, and

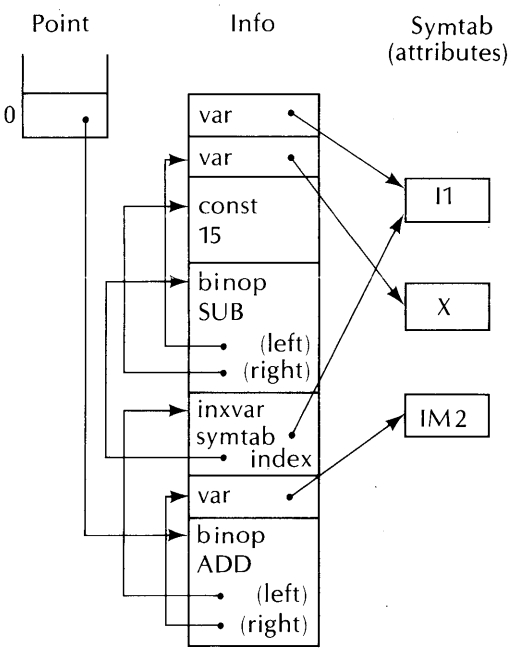


Figure 7.16. Appearance of the generalized translation system (see figure 7.12) after translation of the expression I1(X−15)+IM2.

we then no longer need the tree. The evaluation and removal of the tree is triggered by some other production containing E in its right part, demanding evaluation, e.g.,

$$S \rightarrow V := E$$

a replacement statement. Of course, the "V" is some variable and corresponds to some variable type such as

```
type VARTYPE:
     record case: set of {simple, inx} of
          simple: (SYMPNTR: ↑SYMTAB);
          inx: (SYMPNTR: ↑SYMTAB;
               INDEX: ↑EXPR)
end
```

We let the "S" data structure be empty, so that upon completing this production, the INFO stack will be emptied as well. The following program illustrates a recursive top-down AST evaluator. It generates stack code. The target machine has a separate index register whose value is first developed on the stack, then placed in the index register by STAX. The instruction LA is a "load address"; it determines the address of the following operand and pushes

it. TOS refers to the stack top; TOS − 1 to the operand just below the stack top operand.

```
var S2=↑VARTYPE;  {pointer to VARTYPE}
S2:=POINT[2];

if S2↑.CODE = inx then
begin {left member is indexed variable}
   EVALUATE(S2↑.INDEX);   {get index value on stack}
   EMIT('STAX;');   {code to push into X register}
   EMIT('LA ');
   EMIT(S2↑.SYMPNTR.ADDR);  {address connected to vari-
                             able}
   EMIT(',X;');   {indexed, and closing semicolon}
end
else
begin  {simple variable}
   EMIT('LA ');
   EMIT(S2↑.SYMPNTR.ADDR);  {address connnected to vari-
                             able}
   EMIT(';');   {the closing semicolon}
end;

EVALUATE(POINT[0]);   {evaluate right member expression}
EMIT('STOR TOS-1,I;');   {do the store through stack label}
EMIT('DEL;');  {drop the label left on stack}
REPLACE;   {empty replaces all the stuff used in these
            productions}
```

Exercises

1. Develop a set of productions, a set of data structures and apply actions to generate an AST of the form shown in figure 7.10, for an array reference or procedure call. Note that the "A-ref" node has an indefinite number of children. It must therefore be written on the INFO stack after each of the expansion trees e1, e2, ... are written. *Hint*: write right-recursive productions for the expression list, and examine the sequence of bottom-up apply actions for such productions.

2. Develop productions and data structures for a CASE statement whose AST has the form shown in figure 7.11.

3. Consider the AST evaluator given at the end of section 7.5.3.

(a) Give the AST and the emitted assemble code for each of the following replacement statements:

$$A := B + 3*(C - D);$$

$$X[I-J] := Y[K] - 15;$$

(b) Note that the address of the left-hand member of a replacement statement is pushed in the run-time stack before the right-hand member is evaluated. Suppose instead the index of the left-hand member could be evaluated after the right-hand member is evaluated. How could this be done?

(c) Instead of pushing the address of R in a nonindexed replacement statement R := E, it would be more efficient to simply emit "STOR R" after evaluating E. Show how to modify the evaluator program to do this.

4. Develop a system that builds a linear linked list of INFO elements, given (a) right-recursive productions and (b) left-recursive productions. What is the order of the linked list relative to the order of the source list in each case?

7.6. Bibliographic Notes

The notion of an SDTS was first formalized by Lewis and Stearns (Lewis [1968]). The top-down and bottom-up transducers are adapted from Aho and Ullman (Aho [1972a]), chapter 3. A general discussion of tree generation and tree transformations may be found in DeRemer [1974a].

A great deal of literature exists on automatic translator writing systems. An old, but comprehensive, review is in Feldman and Gries' paper (Feldman [1968]). A more recent survey is given by Griffiths [1974c]. Specific syntax-directed translators are described in Koster [1974], Brooker and Morris (Brooker [1963]), and Metcalfe [1964].

CHAPTER 8

STATIC REPRESENTATIONS OF DATA OBJECTS

Every common programming language permits a programmer to invent names for the entities that he wishes to have the computer manipulate when his program is executed. Properties may be assigned to his names through special language forms called *declarations*. These properties can then be used to define a particular class of operations for the named entities. For example, an addition operator "+" can be used for a variety of different kinds of addition—real, vector, fixed point—by assigning a type to its operands through declarations. In this way, a few operators can be used for a large number of related mathematical operations, increasing the power and clarity of the language.

Some languages permit rather arbitrary aggregates of data to be associated with a single name, for example, a matrix, a tree, or a linked list. The elements of the aggregate can be accessed through various devices, such as indexing or the use of composite names.

This chapter deals with the static association of user-defined names with attributes. By *static*, we mean those associations that the compiler must keep track of, as opposed to those that must exist when the compiled program is executed. There are several aspects to static name association:

- Names will appear in the source more or less at random. Their association with a set of attributes requires some means of efficiently locating a particular name in a table.
- Several languages provide static scoping of user names through blocks. A name will be useful only within its block. The same name may appear in different blocks, but will represent different entities.
- A compiler may operate through one or several passes through the source or through an abstract syntax tree generated on the first pass. In general, the attributes of the user names are needed in all the passes.
- If the source code is free of errors, the names are no longer needed once their attributes have been established and linked to the AST. However, the compiler must be prepared to deal with source errors, and error messages are often enhanced by including appropriate names.

8.1. Symbols, Declarations and Uses

Different kinds of symbols appear in most source languages. There are symbols such as +, *, (,) that carry at most one or two specific, fixed meanings in the language. There are strings of characters that represent constants, such as quoted strings and numbers, called *literals*. Finally, there is a class of user-defined names, called *identifiers* that carry no inherent meaning, but rather are assigned meaning through their context within the source program.

An identifier is typically a string starting with a letter and containing only letters and digits. It may have a bounded length (one to seven characters in Fortran, for example), or unbounded. The strings in the source that comprise identifiers are usually distinguished by a lexical analyzer. As we have seen in chapter 3, the task of distinguishing identifiers from reserved words in certain languages is not easy.

An identifier can be associated with any of a number of entities in a programming language, for example:

- As a reference to some data area. The area may contain a simple fixed datum or several data associated in some fashion.
- As a component of a *composite name*. The composite name may refer to a data area, but the components refer only to some subset of the data area. Composite names consist of a sequence of identifiers separated by some special separator symbol, e.g. COMPANY_PAYROLL in Cobol.
- As a reference to a statement location in a program, i.e., a *statement label*. We shall use the term *label* in this sense hereafter.
- As a procedure name.
- As a macro name.
- As a procedure or macro parameter.
- As a file, or a program, or a device connected to an input-output port of the computer.

8.1.1. Attributes, References and Declarations

The set of meanings associated with an identifier is called its *attributes*. Some examples of attributes are:

- Whether the identifier represents data, a procedure, a statement label, a file, a macro, etc.
- If it represents a datum, which of several possible kinds of values it may take on, its location, and its links to related data.

- If it represents a procedure, the location of the procedure, the number and kinds of its parameters, whether it is user or system defined, whether it can be called by a user or not, etc.
- If it represents a file, the characteristics of the file–record size, whether fixed or variable length, whether sequential, random access, etc.

Clearly, attributes require some kind of classification scheme, as each of the different kinds of identifiers in the language require a different form of attribute specification. The number of attributes may also depend on more than one declaration.

The compiler data structure that associates identifiers with their attributes is called a *symbol table*. Often the identifiers are carried in a separate table called a *name table*, with the attributes in a separate table called an *attribute table*. We shall see that in most cases, the symbol table and its associated abstract syntax tree (AST) can be carried in a single push-down stack.

A statement whose principal purpose is to assign attributes to some identifier is called a *declaration*. An identifier is said to be *declared* when it has appeared in a declaration. An identifier is said to be *referenced* or *used* in a statement in which it appears, but in which no attributes are added to the identifier's attribute set. Sometimes, an appearance of an identifier is both a declaration and a use. For example, the Fortran statement

SAM = I+1

in which I appears for the first time in the source, is both a declaration and a use of I. Under the Fortran rules, identifiers beginning with I, J, K, L, M, or N are assumed to be simple integer variables, unless there is a preceding declaration to the contrary.

In general, a declaration usually causes no object code to be emitted, except possibly to allocate space for data. A reference is usually associated with some generated object code. Of course, a macro name reference results in the generation of an expansion source string and may not be acsociated with any object code generation.

An identifier that only appears in declarations and is never referenced is useless–it need never appear at all. An identifier that appears in two or more conflicting declarations in the same static scope is said to be *multiply declared*. An identifier that appears in a reference, bet never in a declaration is said to be *undeclared*. Whether undeclared identifiers are permitted depends on the language–Fortran permits a class of undeclared identifiers, while Pascal does not.

All the declarations of some identifier must usually precede any reference of the identifier. There is no particular implementation reason for this language policy, as a multi-pass compiler can deal with declarations and references in any order. However, it is good programming style to group the declarations together in a section of the source program that precedes all the

executable statements. When declarations must precede references, the compiler can generate completed object code in one pass, except for forward branches.

8.2. Scope of Identifiers

Every identifier possesses a region of validity within the source program, called its *static scope* of definition. An identifier is *available* within its scope and *unavailable* outside its scope. A reference to some datum through its identifier is valid only if the reference lies within the identifier's scope.

The languages Fortran and Basic have very simple scope rules—the scope of any identifier is an entire program or procedure, and a procedure cannot be nested within another procedure.

In a block-structured language, the scope of an identifier is that of the block in which it is declared. However, the same identifier can be declared several times within the same block without conflict. The scope rules may be expressed as:

1. A *block* is a sequence of source with the structure HEAD DECLS STMTS TAIL where HEAD may be the keyword BEGIN or a procedure declaration head, DECLS are some declarations, STMTS are executable statements, and TAIL is usually the keyword END.

2. Two blocks A and B must be disjoint, or one must be contained in the other. If block A is contained in B, A must be among the DECLS or STMTS of B. Block A is then said to be *nested* in block B, and block B is said to *cover* block A.

3. An identifier declared in DECLS in a block A is available throughout block A, with the exception noted in (4) below. It is not available outside block A.

4. An identifier may be declared in a block A and also in a block B nested in A. The same identifier then represents two different objects. They are best distinguished by renaming the B object for each appearance of the identifier in block B. In any case, the A object is unavailable within block B.

5. The same identifier may be declared in two disjoint blocks A and B, and represent different objects. As in (4) above, we may rename the B object to eliminate confusion. Clearly, the A object is available only in block A and the B object only in block B.

These scope rules are of considerable value to programmers. A new block of source may simply be inserted into an existing program. If new temporary variables are needed in the new block, they may be declared within the block without any concern that one of their names may have already been used in a covering or disjoint block.

8.2.1. Single and Multiple Pass Compilers

A compiler is said to be *single-pass* if the object code can be fully generated in one scan of the source. It is said to be *multi-pass* if two or more scans of the source, or a source representation, are needed.

A single pass compiler almost always requires one or more tables or *fixup lists* to achieve its translation. A common problem encountered in a single pass compiler is the resolution of branches. A forward branch instruction must be emitted to the object code list before its target location is known. When the target location is found, the instructions must then be adjusted. Sometimes the adjustment is done by a *loader*, just before the program is executed; the loader then provides some of the services of a two pass compiler. Otherwise, the compiler must carry a list of instructions that are to be adjusted or "fixed up" when certain statements are located.

A multi-pass compiler can build a symbol table and an AST on the first pass, then scan through the AST one or more times to resolve any code or data locations before emitting object code. Some multi-pass compilers actually scan the source a second time, performing all the lexical analysis and symbol table lookups on each pass.

The principal advantage of multiple passes is that each pass can concentrate on just a few aspects of the compilation process. Each pass requires a fairly small program, but one that is substantially different from those of the other passes. A single pass compiler requires only one scan of the source, but must invoke all its translation algorithms and code fixups repeatedly throughout the single source scan.

The combination of multiple passes and block structuring requires a fairly elaborate symbol table structure. We shall examine three general cases—the "single scope" case which can serve for single or multiple passes, the "single pass, multiple scope" case, and the "multiple pass, multiple scope" case.

Single Scope Symbol Table

The symbol table and AST for a language with a single scope of variable names can be carried in one push-down stack, as indicated in figure 8.1. The attributes and names can be carried in a single data element. The same identifier cannot be declared twice, so that names and attributes carry a one-to-one association.

A name declaration requires a search for the name in the symbol table; if already present, we have a multiply declared name—an error or not, depending on the language. A reference also calls for a search to determine its attributes. The AST can be built with pointers as needed to the identifier attributes. Names are entered before any AST references can be built.

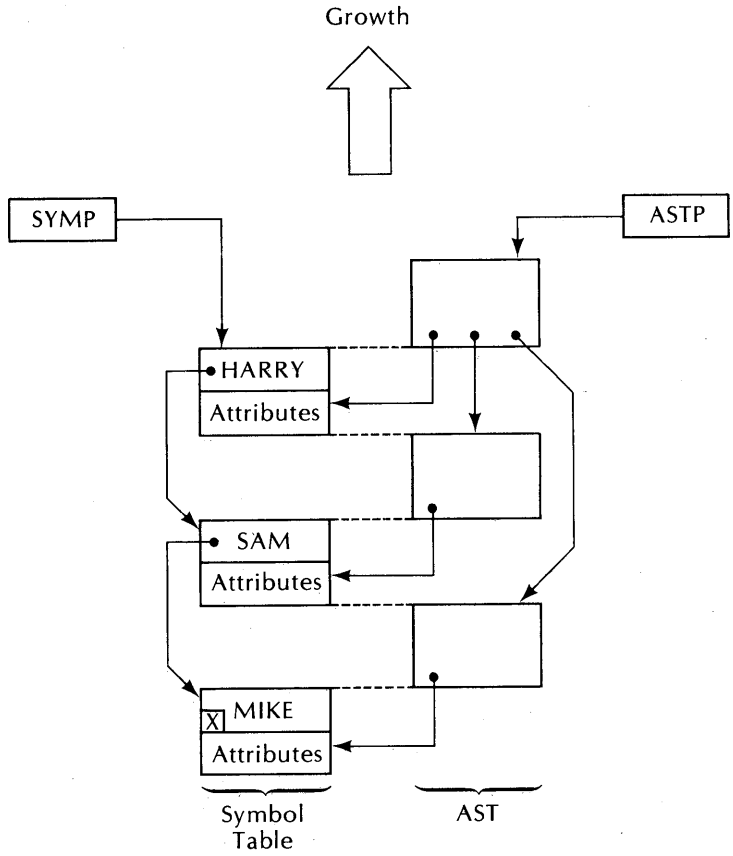


Figure 8.1. A single scope symbol table and abstract-syntax tree (AST). The AST and symbol table can be part of one push-down stack.

Single-pass, Multiple Scope Symbol Table

Again, the names, their attributes, and the AST can share one push-down stack. With multiple scopes, the same identifier can be associated with several different sets of attributes. Some identifiers such as I or N tend to appear again and again in a program. We can therefore achieve some economy of stack space by writing a given identifier in the stack only once, and then referring to it by pointers in the attribute entries. Figure 8.2 shows two scope levels with six data objects and four identifiers currently active. The corresponding program might be the following:

```
BEGIN
  var SAM, FRED, MARY: integer;
    .
    .
    .
  BEGIN
    var FRED, SAM, BILL: integer;
    {Symbol table illustrated for this scope}
      .
      .
      .
  END;
END;
```

The rules for constructing and maintaining this symbol table-AST structure are as follows:

Scope entry. Upon entering a new scope (a BEGIN or PROCEDURE), the stack top is marked (the horizontal dashed line in figure 8.2). This provides demarcations to separate the scopes in the stack.

Scope exit. Upon leaving a scope (through an END), the stack is popped to the top-most mark. The pointers SYMP and ASTP, to the top symbol and the AST root, respectively, are adjusted by following the element linkages down into the stack to just below the mark.

Declaration. Upon a new identifier declaration, a search is made for the name through the attribute pointer chain, starting at the SYMP attribute (stack top). The search should continue to the bottom of the stack, in order to share identifiers, however, the identifier is considered to be multiply defined only if it is found connected to an attribute above the topmost mark. If the identifier is not found, it must be added to the stack. In any case, a new attribute element is also added to the top of the stack.

Reference. Upon a reference of an existing identifier, a search is made for the name through the attribute elements, starting with SYMP. The search ends on the first matching identifier, although other attribute elements deeper in the stack may also point to the same identifier (they must be in covering scopes.) A reference usually results in some addition to the AST; the SYMP pointer is clearly available for this purpose.

At the end of a block, all the attributes, identifiers, and the AST portion associated with the block may be discarded. Of course, this is not done until the AST has been scanned, the block's code emitted, a symbol table map printed, etc.

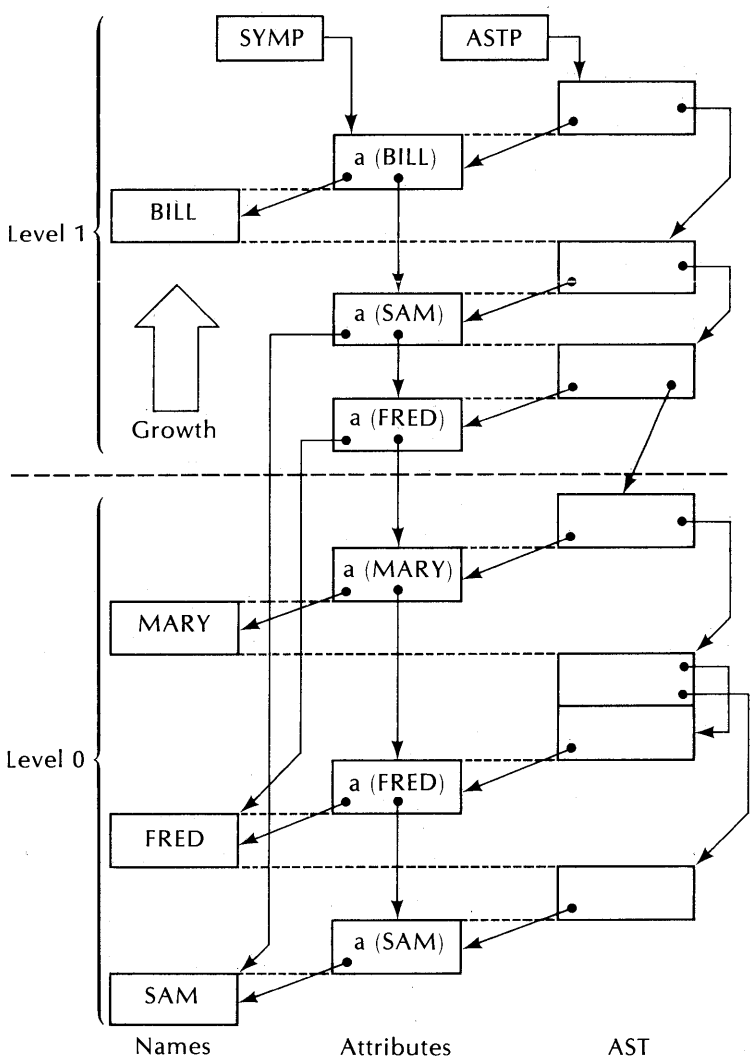


Figure 8.2. A single-pass multiple scope symbol table and AST, with name-sharing. All three structures can share one stack.

Multiple Pass, Multiple Scope Symbol Table

A multiple pass, multiple scope symbol table takes the form shown in figure 8.3. A common name table, with each name appearing exactly once can be used as in figure 8.2. However, the attribute fields must be organized as a tree, and carried through all the passes. The name table, attribute tree, and

AST are built in the first pass. Nothing is discarded at a block end, although some names and attributes might be written to secondary storage to conserve primary storage space.

On the second and subsequent passes, the AST is scanned, and its links to the attributes are used. The name table is unimportant, except as a resource in case program errors are found.

The tree of figure 8.3 corresponds to the following program:

```
BEGIN
  var a1, a2;
  BEGIN
    var a3, a4;
    BEGIN
      var a5, a6, a7;
    END;
    BEGIN
      var a8, a9;
    END
  END;
  BEGIN
    var a10, a11;
    BEGIN
      var a12, a13, a14;
       {SYMP pointer effective here}
    END
  END
END;
```

The attribute tree, name table, and AST may be part of one push-down stack; however, if we choose to discard the name table at the end of the first pass, it must be in a separate stack. A simple mark in the stack is inadequate as a scope delimiter. We need a mark position in each attribute element. In general, SYMP will be NIL or will point to some attribute element. The chain from SYMP down to the stack bottom will then represent a currently active set of identifiers from the innermost nested block to the outermost block. SYMP will move around the tree structure as we move from block to block in the source.

The rules for constructing the attribute tree on the first pass are as follows:

Block entry. Mark the attribute node pointed to by SYMP, then prepare to create a new son of that node.

Block exit. Move pointer SYMP down through the attributes associated with the current level until a marked attribute is found. *Note*—pointer SYMP may already point to a marked node.

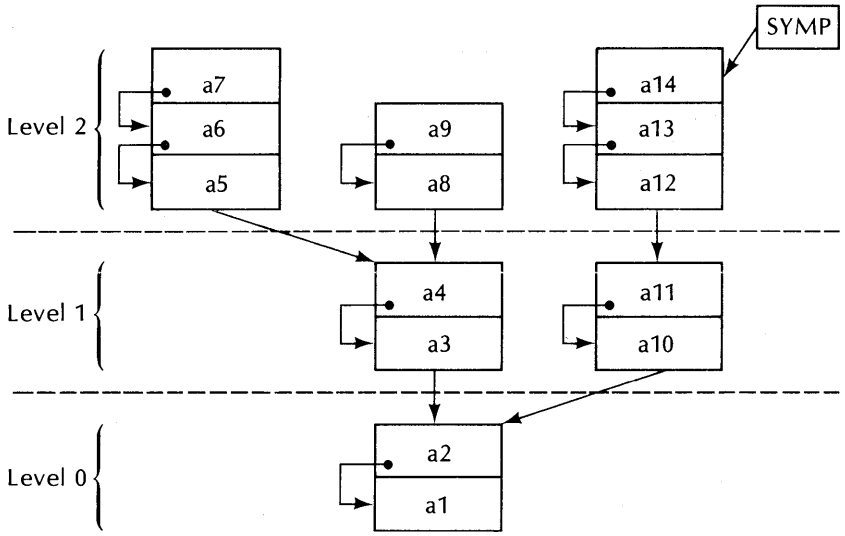


Figure 8.3. A multiple-pass, multiple scope symbol table. The attributes form a tree, with pointers to each other and to a name table. The AST points to the attributes. The structure can be built in one stack.

Declaration. A declaration calls for a new attribute element, to be linked to that pointed to by SYMP. The identifier search is along the tree path from SYMP to the stack bottom, following the attribute chains into the stack. Each attribute element points to some name in a name table (not shown in figure 8.3). A name is multiply declared if found in the current level, but not otherwise. The search should continue to the bottom of the stack if names are to be shared.

Reference. A reference calls for a search along the path from SYMP to the bottom of the stack. The identifier is undeclared if not found somewhere in the tree path.

On the second and subsequent passes, the AST is simply scanned through the links established in the first pass. SYMP is unnecessary.

8.2.2. Algol Statement Labels

In Algol 60, a label is considered declared when it appears as a statement label. Thus a use of a label can precede its declaration. Every other entity must be declared prior to its first use. For example, the following is a legal block in Algol 60:

```
begin
    go to LBL;      {use of LBL}
    I:=I+15;
 LBL: I:=I+1;       {declaration of LBL}
end;
```

The usual scope rules apply to labels—the scope of a label is the innermost block that contains the declaration. Thus in the above program segment, LBL is unavailable outside the begin-end pair.

In order to handle Algol labels, we assign one of the following three attributes to each statement label:

D: declared only (no reference)
RU: referenced at least once, but undeclared
R: declared, and referenced at least once

State D applies to a label that has appeared as a statement label, but not yet in a GO TO. State RU applies to a label that has appeared in a GO TO, but not as a statement label. State R applies to a label that has appeared both ways. These three states permit the compiler to establish, at the end of a block, that:

- A label has appeared exactly once as a statement label
- A label has appeared at least once in a GOTO

The first condition is essential—if it does not hold, an error exists, and should be reported. A label appearing in the symbol table but not as a statement label implies that it has appeared in a GO TO, and that GO TO has no branch target. A label appearing more than once as a statement label is multiply defined.

If the second condition does not hold, a label has appeared only as a statement label, not in a GO TO, and may therefore be removed from the source. Failure of the second condition should be brought to the attention of the programmer, as he apparently had something in mind when he labeled the statement.

We can then implement the label states as follows:

Block entry. Mark the attribute stack as usual.

Block exit. Scan the current stack scope for the attributes RU and D. A "D" attribute can be considered a warning, but the "RU" attribute is an error. (However, there is a complication as we shall see in the next section.)

Statement label. If the label is not in the symbol table, then enter it with attribute D. If it is in the symbol table, it must be a label with attribute RU, R, or D. If R or D, we have a multiply declared label—an error. If RU, this appearance is legal, and the attribute must be changed to R.

Reference. Here, the label has appeared in a GO TO. If the label is not in the symbol table, then enter it with attribute RU. If it is in the symbol table, it must be a label with attribute RU, R, or D. Any of these is legal, however, attribute D is changed to R, while attributes RU and R are unchanged.

These rules apply only on the first pass. On subsequent passes, every label will be marked "R" in an error-free program. Its associated location will be

known only in terms of some node position in the AST. Eventually, the AST will be reduced to minimal form and code will be emitted. Only then will the absolute object code locations of the statement labels be known.

A Complication with Algol Statement Labels

Under the Algol scope rules, a GO TO in some block can refer to a statement label anywhere within a covering block, so that a transfer from within some nesting level can occur to outside the block.

Now consider the following program:

```
{1}  begin  {start of block B1}
{2}    begin  {start of block B2}
{3}      go to L1;
                .
                .
                .
{4}    end;   {end of block B2}
{5}  L1: . . .
{6}  end;     {end of block B1}
```

Upon encountering statement 3, containing the first appearance of the statement label L1, L1 must go into the symbol table. In the absence (so far) of any other information, it would be placed in the block B2 scope. Now at the end of block B2 (statement 4), the declaration of L1 has not yet appeared. Under Algol scope rules, the label need not be in block B2—it can be in some covering block. This causes a certain problem for the compiler upon reaching statement 4. An extension to the block exit mechanism is needed to deal with this situation.

Clearly, the needed extension is some means of moving the L1 attribute from the block B2 scope into the block B1 scope. This may have to be repeated at the end of block B1, if the label L1 has not yet been declared, until the end of the program is reached, if necessary. Only then may the compiler conclude that label L1 is undeclared. In this way, label L1 is "handed down" through the covering blocks until its declaring block is found, or until the end of the program is reached.

Moving an attribute from one scope into another is fairly easy in a single-pass compiler. In figure 8.2, when the end of a block is reached, that block's attributes are discarded. Instead of discarding the undeclared labels, they may simply be kept in the stack, and the stack mark moved to cover them.

In a multi-pass compiler, attribute movement can only be achieved by organizing all the attributes as linked lists. In figure 8.3, the attributes of each block are shown as contiguous, but they must in fact be linked by pointers.

Then the "movement" of an attribute from one block to a covering block is a matter of changing a few pointers. We leave the details to an exercise.

Exercises

1. Sketch a single-pass and a multiple-pass symbol table for the following multiple scope program. Show the table structure at the end of each block, just before the block exit occurs.

```
begin
  var ED, SAM;
  begin
    var MIKE,MARY,SAM;
  end;
  begin
    var ED,MIKE;
    begin
      var SAM,BILL;
    end;
    begin
      var AL,MARY;
    end;
  end;
  var SAM,BILL;
end;
```

2. In the multi-pass, multiple scope symbol table structure of figure 8.3, are reverse links between the attribute elements needed?

3. Consider Fortran data declarations. An identifier may appear in several different declarations in any order without error, as follows:

   - an EQUIVALENCE declaration
   - in a COMMON declaration
   - in a type declaration, e.g. INTEGER SAM,FRED(15)
   - in an ARRAY declaration, e.g. ARRAY MIKE(250)
   - in a DATA declaration

   Furthermore, if an identifier is dimensioned in a type declaration, it should not appear at all in an ARRAY; if it appears in an ARRAY declaration and in a type declaration, it should be dimensioned in only

the ARRAY declaration. Give a set of attributes and rules for dealing with Fortran data identifiers. *Hint*—construct a finite state automaton that describes all legal declarations for an identifier.

4. Show informally that the symbol table structures illustrated in figures 8.2 and 8.3 guarantee that (1) an identifier declared in some block is available only within that block, and that (2) an identifier I declared in block A is unavailable in a nested block B, if I is also declared in B.

5. Suppose that a multiple pass compiler were to be implemented using the structure of figure 8.2. The multiple passes are only through the current scope. That is, as soon as a block end is reached, several more passes through the AST associated with that block are made to finish the compilation. Then the current block's symbol table can be deleted. Could such a system be made to work? Would there be any advantage, if so, over a single pass system?

6. In the single pass scheme of figure 8.2, the names are separated from their attributes in order to reduce the storage for the identifier strings. Suppose that name sharing is not attempted—a name could then be part of its attribute structure. The advantage of this is that a pointer from the attribute to the name is not needed; the disadvantage is that the same name may appear multiple times in the stack. Let an average name length be N bytes, and a pointer be 2 bytes. How much name sharing must occur in order for separated names to require less space than names associated with their attributes?

8.3. Data Objects and Their Static Representation

A *data object* is some entity associated with a name that comes into existence when the program that contains it is executed. It is carried by some memory field for the duration of its life, and it is operated upon by various program instructions. A given object may carry a variety of different values during execution and may reside in physically different sections of memory at different times. Its key property is continuity; its value is preserved between accesses and is only changed through some assignment statement in the program. Its life may or may not end when the program control passes out of its identifier's scope. Some data objects are created initially and survive for the life of the program, while others survive only while program control remains in the static scope of its identifier.

A data object has an external and an internal structure. Externally, its structure is defined by its language declaration—e.g. REAL or BYTE ARRAY. Internally, it is some field of binary information in memory,

possibly scattered about in the available memory space.

To a compiler, a data object is an entry in a symbol table. It possesses a set of attributes, and they may include a prescription of its location in memory during execution. The operations on a data object implied by the source language are partially determined by its attributes and result in a certain sequence of emitted code that, when executed, affect the memory cells assigned to the data object.

The treatment of the data object associated with an identifier depends on the attributes assigned to the object through its declaration. In most computer systems, the internal form of every data object is some sequence of binary digits which provides no clue as to its purpose. It is the task of the compiler to associate meaning with every data object and to guarantee that the specifications of the program with respect to the data object are faithfully reflected in the sequence of machine code being generated.

8.3.1. Primitive Objects

A *primitive data object* is an object that is normally treated as a unit by the operations of the language. It usually cannot be subdivided by any operation, and in many modern languages, the intebnalstructure of the primitive objects is undefined.

Each data object is internally encoded in some manner that depends on the object machine system and the programming language system. For example, the 16-bit binary number

1110010100000010 or (octal) 162402

can be interpreted in any of the following ways on an HP3000 computer:

1. As the memory reference instruction STB Q+2, I (store a byte on the top of stack into a location marked by an indirect address located 2 words relative to the Q register).
2. As the "integer" −6910 (decimal form).
3. As the "logical" 58626 (decimal form).
4. As an indirect address, pointing to a location 6910 words offset from the contents of the program register, "PB".
5. As the pair of bytes (8 bit data objects) 345 (octal) and 2. These bytes may be interpreted as the ASCII characters "c" and "STX", e.g., lower case letter "c" and the special character "STX".

One of these interpretations is selected during execution by an operation on the data. Thus the STB interpretation holds when that word is fetched into the instruction register and executed.

If the word is fetched and loaded in a data register, and then an "integer add" operation is executed, it is effectively interpreted as an "integer." If the

operation is a "logical multiply," then it is interpreted as a "logical," and so forth.

Every computer has a set of definitions of the primitive data objects upon which its instruction set is designed to operate. Matching the computer operations and data objects to those required by a source language is sometimes troublesome. For the sake of efficiency, as many of the machine operations as possible should be used for source language operations; if a difference in specification exists, then the language operation must be simulated by a procedure call or a sequence of machine operations. For most common languages, the common machine operations and language operations are arithmetic in character, making a match possible. However, different machines use different word lengths and representations, which affect the maximum range of numeric values. Some idea of the variety of number representation may be gained from the following examples.

1. On the IBM 1620, numbers are represented as a string of characters, representing decimal digits. Special marks are included for the representation of floating point numbers, signs, and the end of a number, since numbers may be of different lengths.
2. On the IBM 360, numbers may be represented in a variety of forms. There is a set of binary integer forms, 16 and 32 bits in length, short and long binary floating point numbers, and two decimal numbers (packed and unpacked). A small positive integer may be represented as a byte.
3. On the CDC 6000 series, only floating-point numbers 60 bits long are handled by the instruction set. The format is interesting, as there are representations for "infinity" and for "undefined." Integer arithmetic is done with the floating-point hardware. The number format is 1's complement, rather than the usual 2's complement.

The representation of characters as an internal code is similarly nonstandard among the computer manufacturers. IBM has used EBCDIC, many smaller manufacturers use ASCII, and CDC uses its own kind of 6-bit character code, called *display code*, for character representation. Even within one manufacturer and one standard code, some variations are found, for example, ASCII has different versions to reflect differences in foreign languages.

Printers, terminals, and other character-oriented devices also exhibit coding variations, even within a single manufacturer's product line.

Now consider the source language. What are its specifications with regard to its primitive data objects? Let us consider some example source specifications.

1. ANS FORTRAN (Campbell [1976]) says this of the "integer type" and "real type":

"An integer datum is always an exact representation of an integer value. It may assume a positive, negative, or zero value. It may assume only an integral value. An integer datum has one noncharacter storage unit in a storage sequence. A real datum is a processor approximation to the value of a real number. It may assume a positive, negative, or zero value. A real datum has one noncharacter storage unit in a storage sequence."

Of characters, the Fortran standard states "a character datum is a string of characters," and that "a character datum has one character storage unit in a storage sequence for each character in the string." The internal form of a character is left unspecified. The standard only addresses the issue of connectedness of a set of characters that constitutes a string.

2. In Algol 60 (Naur [1963]), three primitive objects (integer, real, Boolean) are provided. Very little is said about them except that integers and reals may be positive or negative numbers, and that Booleans may only assume the values TRUE and FALSE.

3. Pascal recognizes four primitive objects: integer, real, char, and Boolean. The report says of integer, for example, only that "it is an element of the implementation-defined subset of whole numbers."

We see that the structure of the primitive data objects are not specified in these languages. Each of these languagec is intended to be implemented on a variety of computers. The languages are also such that the internal details of primitive objects are not essential to an algorithm and should not be. If nothing in a program depends on the implementation details, then the program can very likely be transported to several quite different computer systems.

However, a compiler is implemented on some machine and has a particular object machine. It is usually desirable that the compiler, too, be as nearly machine-independent as possible, though it cannot entirely be so. The characteristics of primitive data objects enter a compiler in two ways: (1) some choice of data type must be made to handle conversion of source language constants, and (2) some section of the compiler must deal with conversion (if any) from a compiler "internal" form of constant to the object machine data form. Both these tasks clearly should be relegated to a few low-level procedures that can be easily modified if necessary for the sake of portability. The bulk of the compiler algorithm should be indifferent to the internal representation details demanded by any one host machine.

The permissible ranges of primitive data objects are subject to several limits: (1) those imposed by the source language (usually there are none), (2) those imposed by the compiler implementation (there should be none), and (3) those imposed by the object machine. Of these limitations, the last is the most crucial for performance. As many operations as feasible should exploit the machine hardware provided for the purpose, and these operations may function slightly differently on different machines.

Some languages provide means of specifying minimum number ranges or number properties. In Algol 68, a program may be preceded by a *standard prefix* that contains a minimum set of specifications that must be met by the operators and data objects of the program. An Algol 68 compiler for some target machine is expected to check this specification set against the target machine and report any difficulty in meeting the specification.

In Pascal, numeric data may be specified with a range, e.g.,

```
var day: 1..31; month: 1..12; year: 0..2000
```

The judicious use of a range, rather than just "integer," can overcome potential problems with numeric ranges and also make possible reductions in the size of the data structures.

In PL/I, the user may specify a base (decimal or binary), a scale (number of places by which a decimal point is assumed shifted), a precision (number of significant digits), and a mode (type). All of these ranges may be specified in a declaration, and the compiler is expected to select a target machine data object that meets or exceeds the source specification, and to perform whatever special operations that are necessary to make the machine data object operations conform to the source specifications.

Fortran provides no number range specification. Worse, standard Fortran permits quite general equivalences of one data object to another, e.g.,

```
DOUBLE D;
REAL R;
EQUIVALENCE (D, R);
```

With these, the data space associated with the double D and the real R are shared. Clearly, a determined (or naive) programmer can access the internal structure of a REAL (very machine dependent) in an otherwise high-level program.

8.3.2. Types

A *type* is a means within the source language of assigning a set of attributes to a data object. Most languages provide a set of standard primitive types, and some permit the user to organize a set of types and then to assign a type name to that set.

The assignment of a type to a data object is made through its declaration. Typing provides several benefits of considerable value:

1. The user often has a choice of more than one applicable type. He may then choose one that represents the best compromise between a needed range and memory economy.

2. A few operator symbols can represent a large number of operations. For example, the symbol "+" can represent the addition of a large variety of numeric objects, including pairs of differently typed objects. The specific operation implied by a "+" then depends on the types of its operands.

3. The compiler can provide a strong guarantee of operation integrity through the use of internal data conversions, warning messages, or error messages.

A *strongly-typed* language is one in which a typing of every object is required and is rigorously enforced. A compiler for such a language will never permit an object of some type to be operated upon as though it had another type, without an explicit command in the source language, or without a conversion.

The controls provided through data object typing are defeated if any of the following language features exist and are used:

1. Equivalences. An equivalence permits some target machine data object field to be shared among two or more objects, with possibly different types.

In Fortran, an equivalence can be created through the EQUIVALENCE statement or through the COMMON statement. Two different data objects thereby share a common data space. An equivalence of a Fortran REAL and an INTEGER will clearly permit some kind of access of the bits in the REAL as though they represented an INTEGER instead, without numeric conversion. Any program that exploits such a capability must have been written with a particular implementation in mind and is not transportable.

2. Binary operations on nonbinary types. For example, if the language permits a shift or a bit-by-bit logical operation on a real number, the real can be altered or accessed in ways that depend on its internal representation.

3. The use of characters or other nonarithmetic objects in arithmetic contexts. For example, the conversion of a string of digit characters to its equivalent numeric value requires a translation of each digit character to its equivalent value. It is best that this be done through a collation function, e.g., the Pascal function *ord(c).* The function *ord(c)* returns an ordinal collation index for a character c, which is not necessarily the internal code for the character but which represents its alphabetic or numeric ordering.

4. Passing a data object by reference to a procedure call, when its type and the type of the corresponding formal object differ, without also passing type conversion information. Without conversion information, such a parameter pass can only be treated as an equivalence by the compiler.

5. Use of assembly language intermixed with the high level language.

Assembly language enforces no typing rules, and can be used to defeat any compiler system, but at the cost of total nonportability.

6. Writing a file with one format specification and reading it with another. The only protection against this possibility lies in a file label that carries type specifications for the file and a source language that permits only labeled files to be written or read.

7. Use of illegally large or small array indices to access some memory area not assigned to the array by its declaration. This very bad practice is in fact possible with most compilers, because array bounds checking can only be done with additional code, degrading performance. The deliberate use of an illegal array index must be condemned for any but the most primitive, machine-level programs. It presupposes that the adjacently accessed data areas will remain in place in the source, that anyone modifying the source code will understand and respect the practice, and finally that all future versions of the compiler will continue to allocate data object space in exactly the way the programmer expected. These suppositions are rarely true.

8. Uncontrolled use of pointers. A pointer is effectively a memory address. Its address value may be altered during execution and then used to access the information it points to. Pointers are valuable in assembly language, but have not found their way into many high-level languages, because of the difficulty of controlling their use.

Pascal is a notable exception since a pointer is always typed. It can only be declared in association with some type. When it is set, it may only be caused to point to a data object of its own type. Then when the pointer is used indirectly (i.e., to access the data object to which it points), the compiler and the programmer have confidence that the typing rules are not violated.

Type Conversion

A typical problem that arises in typed languages is mixed-mode type conversion, illustrated by the following Fortran example:

$$C = I+J*(R/D)$$

where I and J are type INTEGER, C is type COMPLEX, R is type REAL, and D is type DOUBLE PRECISION (floating point). The Fortran conversion rules are essentially expressed by a type hierarchy as follows:

$$\text{COMPLEX} > \text{DOUBLE} > \text{REAL} > \text{INTEGER}$$

This means that, given an operation on data objects of two types (t_1, t_2), that type t with the higher hierarchy is chosen as the type of the operation. The data object with the lower type is first converted to type t, the operation is performed in type t, and the resulting expression is considered to be of type t.

The type hierarchy in Fortran reflects an ordering imposed by the domains of the various types. Although the Fortran specification says only that an INTEGER or REAL datum has "one noncharacter storage unit," and that a DOUBLE PRECISION or COMPLEX have "two noncharacter storage units" in consecutive order, it is reasonable to suppose that on most implementations,

- Every INTEGER value can be represented by some REAL, perhaps with some loss of precision for large integers.
- Every REAL value can be accurately represented by some DOUBLE.
- Every DOUBLE can be represented by some COMPLEX, perhaps with some loss of precision.

The hierarchy therefore provides a reasonable assurance that any binary operation allowed by the Fortran language will yield a valid result, although with some loss of precision in certain cases.

One binary operation is prohibited–between a DOUBLE-type and a COMPLEX-type data object.

The hierarchy is applied to each binary operation as it appears in an abstract-syntax tree as follows:

Let h(t) be the hierarchy index of a type t, as expressed by the table:

type t	h(t)
INTEGER	1
REAL	2
DOUBLE PRECISION	3
COMPLEX	4

When the AST is constructed for some expression, each leaf is tagged with h(t), where t is the type associated with the leaf. Then each operator node N is tagged with index $h' = MAX(h_1, h_2)$, where h_1 and h_2 are the hierarchy indices of N's children.

For example, a node with a DOUBLE child and an INTEGER child would be marked DOUBLE.

The tree operations implied by these typing rules are of a bottom-up character; we say that the attributes for any node are *synthesized* from its descendant attributes. Clearly, the necessary node types can be constructed while the AST is being built by a bottom-up parser, assuming, of course, that all the leaf attributes are known during the expression parse (they are in most languages.)

An assignment statement in Fortran has somewhat different typing rules. Given the statement

$$v = e$$

a conversion from the "e" type to the "v" type is to be automatically generated by the compiler. A conversion is specified for every pair of types. The conversion from a smaller to a larger hierarchy (e.g., INTEGER to DOUBLE) preserves arithmetic value, though with some loss of precision in certain cases. However, those conversions from a larger to a smaller hierarchy may result in overflow or other undesirable side-effects.

Conversion is easily added to the AST in preparation for code generation. We merely add a unary conversion node between every pair of linked nodes of different type. This is most conveniently done during the bottom-up type decoration of the tree nodes. The assignment node is given the type of its left member, so that a conversion node (if required) will appear between the assignment node and its right member.

The assignment statement given above therefore yields the AST shown in figure 8.4, which carries four conversions. All the binary operations in the right-hand expression are performed in double precision, because of the structure of the expression. If the "*" were to be replaced by a "-", then the I + J would be performed in integer mode and converted to double precision for the subtraction.

The Fortran standard rules illustrate a typing system used in several languages. Other typing systems have been defined for special languages. For example, a Fortran implementation on a CDC 6000 series system has the following typing rule for expressions:

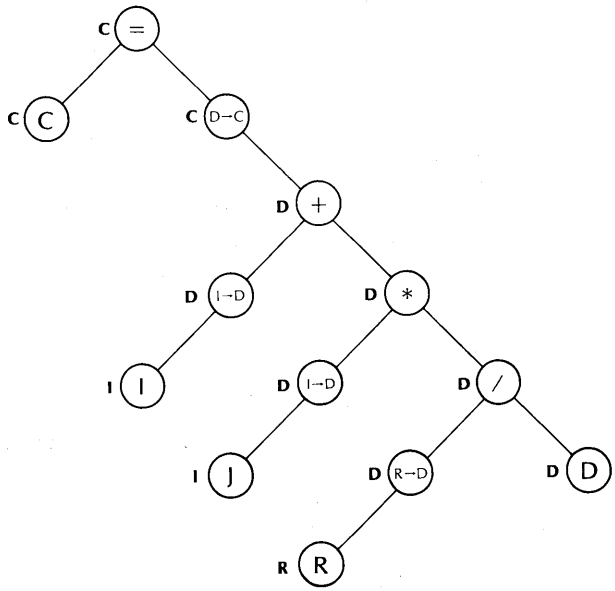


Figure 8.4. A typed AST, with conversions added, for the Fortran assignment statement C=I+J*(R/D). The notation R → D means "real-to-double conversion."

Every operation covered by a pair of parentheses, or every operation of an open expression, is performed in the same type. That type is hierarchically the largest of the types in the set of operands associated with the operators.

A pair of operations is said to be *covered* by a parenthesis pair if the parentheses are matched, and they are either not separated by any parentheses or are separated by a matching pair of parentheses. An *open* expression is one not enclosed by any parentheses.

Thus in the following expression all the + operators are executed in type REAL:

$$(I + I - (R*I) + I)$$

The REAL expression in the inner parentheses causes all the I operations to be performed in type REAL.

We leave as an exercise the construction of a type-decorated AST with inserted conversions for these typing rules.

Exercises

1. Construct a typed AST for each of the following Fortran assignment statements. I, R, D, and C refer to data objects of the four Fortran types. Which of them contains a typing error?

```
R = I*C - COMPLEX(D/I)
C = I*C + I/R*(D+R)
D = R + (I*(I-I))
I = C + C - I*R/I + R
```

2. Give an algorithm for the construction of a type-decorated AST for the CDC 6000 Fortran rules. Notice that parentheses disappear in the AST—what can be done to retain their influence on the node typing?
3. Give an example of an expression that will evaluate differently under ANS Fortran and CDC Fortran typing rules.

8.3.3. Structures

User-defined structures provide powerful means of organizing algorithms. The design and expression of a suitable data structure is as important as the design and expression of the program statements.

A *structure* is any assemblage of primitive data objects organized in such a manner as to facilitate the coding of some algorithm. As a program might be structured through the use of if-then-else or while-do statements and

begin-end blocks, so might data be similarly organized. An optimally clear and concise program results from the balanced use of control and data structures.

The most primitive assemblage of data is probably the one-dimensional *array,* or *vector.* An array permits the selection of any one object of a set by means of an integer index. Each of the array elements have the same type and are normally arranged in contiguous memory locations. Whether an array must be arranged contiguously depends on the language. Fortran permits rather general equivalences, hence the arrangement of array elements in memory is an important language specification. PL/I guarantees contiguity only if the array is part of a "structure" and not otherwise.

An obvious extension of an array is a *multidimensioned array,* or *matrix.* At one time, Fortran permitted matrices with no more than three dimensions. In Fortran IV, an array may carry any number of dimensions. Algol 60 permits arrays of any number of dimensions, as does Pascal and PL/I.

Arrays may be statically or dynamically dimensioned. A statically dimensioned array carries bounds fixed at compile time. A dynamically dimensioned array carries bounds that may be dynamically adjusted when control passes through its declaration, or when a special allocate statement is executed. Dynamic arrays are usually more costly to implement than static arrays, but permit greater control over the use of memory resources and greater flexibility in programming.

A more general data structure is the PL/I *structure,* which is a a tree of data objects. Each of the objects is a tree leaf and may be accessed through a special name convention from the root down. The nodes may be arrays. The purpose of a PL/I structure, beyond documentation (the separate pieces could just as well be declared separately), is to collect disparate data objects together in a contiguous section of memory and to permit all or some of the data objects in a PL/I structure to be accessed as a unit through one name reference.

Another form of data structure is the Pascal *user-defined type.* Here, a data structure may be given a name without an accompanying allocation of data objects. That name may then be used in its own structure, or in other structures, and will ultimately represent some set of data objects. For example, a type that contains its own name can express a quite general directed graph in a compact form.

The issues of data structuring as they apply to compiler design are these:

1. How might a data structure's static properties be represented internally as a symbol table structure?
2. Given a reference to some data object, perhaps a member of some structure, how can its symbol table entry be located? As we shall see, this is not a trivial problem for PL/I and Cobol.
3. How might a data structure be represented at run-time?

4. Given a data structure's representation, how is a data object best accessed?
5. If the data structure is dynamic, what run-time representations and access mechanisms are required? Several different kinds of dynamic structures have been proposed in different languages, and they require rather different run-time representations.

We shall deal with the static issues (1,2) in this section, and deal with some of the dynamic issues (3,4,5) in the next chapter.

8.3.3.1. Array Objects

The most common aggregate of primitive data objects is the array. Arrays are accessed through some base address and one or more integer indices. Whether an array is physically in contiguous memory is important in Fortran, but not in Algol, Pascal, or PL/I. It is usually only necessary that a mapping from a set of indices to a unique element exists. However, we shall develop two access methods; the first requires contiguity, and the second requires contiguity only of each slice of a multidimensional array.

An array may be static or dynamic. A *static* array has a set of bounds fixed at compile time. These bounds are carried in the symbol table as part of the array's attributes, and used to generate code to compute an effective index for every reference. We shall deal only with static arrays here. Dynamic arrays are treated in chapter 9.

Suppose an array is dimensioned

var A: array[$b_1..u_1$, $b_2..u_2$, $b_3..u_3$] of integer;

and we have a reference

$$A[e_1, e_2, e_3]$$

The b_i and u_i are compiler constants. The e_i are expressions to be computed at run-time in general. Now the integer values of e_1, e_2, and e_3 must be within the dimensions, i.e.,

$$b_1 \leq e_1 \leq u_1$$

$$b_2 \leq e_2 \leq u_2$$

$$b_3 \leq e_3 \leq u_3$$

A reasonable compiler assumption is that these constraints are always met. If desired, the compiler may emit additional run-time code to check each of the array reference indices e_1, e_2, $e_3$against the required limits.

The array A may be mapped into linear memory in several different ways. We present two here, suitable for static arrays, and a third, for a dynamic array, in the next chapter.

Linear Mapping. The array A may be mapped onto a contiguous linear memory space in increasing order as follows:

$A[b_1, b_2, b_3]$
$A[b_1, b_2, b_3+1]$
.
.
.
$A[b_1, b_2, u_3]$
$A[b_1, b_2+1, b_3]$
$A[b_1, b_2+1, b_3+1]$
.
.
.
$A[b_1, u_2, u_3]$
$A[b_1+1, b_2, b_3]$
.
.
.
$A[u_1, u_2, u_3]$

Let the address of the first of these be denoted B. Then the offset X of any other element, with indices e_1, e_2, e_3, is clearly

$$X = n_3*n_2*(e_1-b_1) + n_3*(e_2-b_2) + e_3-b_3$$

where

$n_2 = u_2 - b_2 + 1$, and
$n_3 = u_3 - b_3 + 1$.

Note that $n_1 = u_1 - b_1 + 1$ is unnecessary and that X may be rewritten as

$$X = n_3*(n_2*e_1 + e_2) + e_3 - (n_3*(n_2*b_1 + b_2) + b_3)$$

The last term is a compile-time constant, while the first two must be evaluated at run-time. This expression for X suggests that we should

1. Create a fictitious "base" address

$$B' = B - (n_3*(n_2*b_1 + b_2) + b_3)$$

2. On a compiler reference, compute an index

$$X' = n_3*(n_2*e_1 + e_2) + e_3$$

(two additions and two multiplications).

3. Access the element as

$$<\text{operation}> B', X'$$

i.e., an indexed reference through a base address B′.

The generalization of this scheme to any number of dimensions should be clear.

The symbol table attributes associated with A are also clear—we need only the fictitious base address B′ and the parameters n_2, n_3.

Exercises

1. Generalize the linear access scheme to n dimensions, where $n \geq 1$.

2. Matrix elements are usually accessed in a loop, with index values incremented in unit steps on each pass through the loop. Discuss informally some optimizations that could profitably be applied to such loops. For a case study, consider Warshall's algorithm given in chapter 5, or matrix multiplication.

3. Suppose the target machine is a stack machine. Design instructions that carry out as much of the array access as possible, leaving a minimum to be executed by compiler-generated instructions. Are fewer instructions required?

4. Discuss array bounds checking at two levels—(a) each index must be in range of its own dimensions, and (b) the final referenced datum must be in range of the A matrix values. Are there sound reasons for preferring the first despite its obviously higher cost?

5. Suppose that indexing arithmetic is limited to a finite word size, e.g. 12 bits. Discuss the problem of trapping all illegal indexing operations at compile time, given various kinds of lower and upper dimension limits. Is it possible that every dimension and reference is within range of the number size, but that some intermediate result is out of range?

6. Suppose one or more of the index expressions e_i are compiler constants. How might the number of operations required to produce an indexed reference be reduced?

Matrix Pointers. Another way of accessing a multidimensioned array is through a system of matrix pointers. For example, consider a two-dimensional array

```
var X: array [0..m, 0..n] of integer;
```

We construct a vector V of size m + 1, containing pointers. Each pointer is directed to a different row (second dimension) of the two-dimensional array. Then the access of an element Z[i, j] is a matter of fetching V(i), which points to an array slice, then indexing into this slice through index j. On certain machines, this combined operation can be very fast—we only need a set of index registers and a special operation that steps the machine through the pointer arrays to the final data array element.

The fetching of Z[i, j] might appear as follows in stack machine code:

```
LOAD i;    {evaluate first index}
STAX;      {move value to index register; TOS is deleted}
LOAD V,X;  {fetch pointer to the i'th slice}
LOAD j;    {evaluate second index}
STAX;      {move to the index register}
LOAD TOS,I,X;  {fetch the value indirectly through
          pointer on stack and index j}
```

Exercises

1. Develop a formula for the memory space required for an array of n dimensions $d_1, d_2, \ldots, d_n$. Assume that each pointer requires one unit of storage and that each data element requires b units of storage. Then express the percentage overhead required for the pointers and show that this relative overhead decreases to zero as the dimensions increase. (The parameter b is usually 1 or 2.)
2. Discuss the relative efficiency of the matrix pointer and linear access methods, with and without optimizations. Examine a specific algorithm, e.g., Warshall's algorithm or matrix multiplication.
3. Assume the target machine is a stack machine. Design high-level instructions to retrieve and store a data object organized by matrix pointers. Should a pointer have any special flags, e.g., indirection?

8.3.3.2. PL/I Structures

Several modern languages provide means of declaring collections of data organized as lists and/or trees. One of the earlier aggregrate declaration forms is the PL/I structure, illustrated next. (Cobol has a similar structure definition.)

```
DECL 1 PAYROLL(75),
       2 NAME,
          3 LAST CHARACTER(12),
```

```
      53 FIRST CHARACTER(8),
       3 MIDDLE CHARACTER(1),
    2 PAY_NO CHARACTER(5),
    2 HRS,
       3 REGULAR FIXED DECIMAL(2),
       3 OVTIM FIXED DECIMAL(2),
    2 RATE,
       3 STRATE FIXED DECIMAL(3,2),
       3 OVRTIM FIXED DECIMAL(3,2);
```

This structure defines a payroll file for a firm with 75 employees. We have a static tree structure with the leaves: LAST, FIRST, MIDDLE, PAY_NO, REGULAR, OVTIM, STRATE, and OVRTIM. The remaining names are internal nodes. Each of the leaves is an elementary data object or an array. Each of the interior nodes can be an array, and if so, means that its subtree is replicated in memory by as many elements as specified in the array dimension.

The integer prefixes define the *level number*; the syntax does not otherwise fix the tree level numbers.

A data element is accessed through a composite name, e.g.,

```
PAYROLL(15).HRS.OVTIM
```

or

```
PAYROLL(P).NAME.LAST(X)
```

The latter name designates character X of the last name of employee P.

A set of data objects may also be accessed through the name of some internal node in the structure. For example, PAYROLL(2).NAME stands for the set of three data objects LAST, FIRST, and MIDDLE of the second employee.

The name

```
NAME.HRS
```

is illegal; the two parts NAME and HRS are drawn from the same level.

PL/I permits abbreviating a structure element name, by omitting any of its components, provided that the specified data object is unambiguously designated. Abbreviation poses an interesting and nontrivial problem of symbol table access, and poses a need of detecting ambiguous name references, which we shall deal with shortly. Array indices may also be moved to the right, but may not be reordered or omitted. Because of index movement, every array reference must carry an index. For example,

```
PAYROLL.NAME.LAST(P)(X)
```

and

LAST(P)(X)

are also legal representations for the name PAYROLL(P).NAME.LAST(X)

.

The remaining features in the PL/I PAYROLL structure given above are attributes of the leaf node data object:

- CHARACTER(12) is an array of 12 characters.
- DECIMAL(2) is a decimal number with 2 significant places, e.g., any number in the range 00-99.
- DECIMAL(3, 2) is a decimal number with a decimal fraction, 3 digits ahead of and 2 behind the decimal point, e.g., 345.67.

A picture of part of this structure as it might appear in memory at runtime is given in figure 8.5. Only one of the 75 PAYROLL subtrees (the one for index 5) is broken out and completely displayed. For example,

PAYROLL(5).HRS.REGULAR

has the value 40 according to this structure.

PL/I Name Scanner. A PL/I structure name may be completely or partially specified. Any partial specification (consisting of only some of the names along a path from a root to the data object leaf) is legal if a unique data object is thereby specified.

A name in PL/I, whether partial or full, is considered ambiguous only if, as

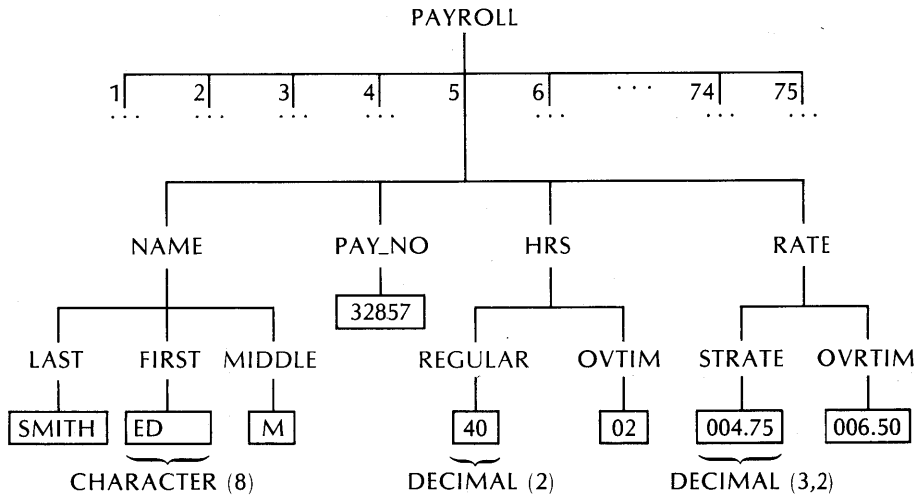


Figure 8.5. A PL/I structure displayed as it is organized at run-time.

a full name, it refers to two or more data objects. (Note that a "data object" may be a set of primitive objects in a structure.) If some name N can refer to two different objects, as a full name for one object A and as a partial name for another object B, then it is assumed to refer to object A, that is, a full name overrides any possible partial names.

It is also possible that the structures of a PL/I program are such that some data objects cannot be unambiguously accessed. If no reference to one of these data objects exists in the program, then the structures are acceptable, despite their inherent ambiguity. We see that any structure is permissible per se; however, each of the name references must be examined for possible ambiguities.

A clever solution to this name abbreviation problem was proposed by Gates and Poplawski (Gates [1973]). It is an interesting application of finite-state automaton methods.

A FSA is constructed with these properties:

1. Each transition is on a primitive name that appears somewhere in a PL/I declaration. For example, NAME is a primitive name in the structure example given above.
2. The halt states indicate either legal or illegal composite symbols. An illegal composite name is ambiguous—it could refer to two or more different data objects, possibly in different structures.

The construction of the FSA is along the following lines. Details and a worked-out example may be found in Gates' paper; also see Abrahams [1974] for additional remarks. Knuth [1968] also gives a Cobol name resolution algorithm.

1. Construct a list of partial names. Each of these is a composite name formed by starting at the root of some structure and following a path part way toward a leaf. Each name contains at least one primitive name. The full names are in this list.

2. Associate a state with each name in the list. Add a unique start state.

3. Each transition in the FSA is on a primitive name. Each sequence of transitions beginning with the start state and ending in a halt state must spell out a fully qualified name. For example, if a full name is A.B.C, then there will be a sequence of transitions:

from	on name	to
S	A	A
A	B	A.B
A.B	C	A.B.C

Thus recognition of the full name A.B.C ends in the state associated with that name.

Clearly, if no ambiguities exist and all names are fully specified, this machine will accept the structure names of the language. However, these conditions are not always met.

4. To deal with partial names, we consider one path P in the machine from the start state S to a halt state. Add all the possible transitions permitted by the following rules:

(a) A transition from state Q to R will be added if a directed path P′ from Q to R already exists, and if P′ is in P.
(b) A transition to a state with name ....X will carry the primitive name X.

Also make every state other than S a "new" halt state.

5. We have constructed a machine that is possibly nondeterministic. Resolve its nondeterminism by introducing composite states. Assign new numbers to the composite states. These states are to be distinguished from the existing states.

6. Repeat steps 4 and 5 for all the machine paths to yield a new deterministic FSA. It will have "old" halt states and "new" halt states. The "new" halt states represent sets of old and/or new states. This machine will accept any abbreviated composite name compatible with the declared structures. If it ends in a "new" halt state, the name may be ambiguous; there could be two or more distinct data that could be so designated (see rule 7). If it ends in an "old" halt state, the name is unambiguous; furthermore, the state is associated with the complete data object name.

7. A "new" halt state represents an ambiguous name if it contains two or more legal full names.

Of course, if no transition on some primitive name exists in the machine, then the composite name cannot be a part of any data structure and is thereby illegal.

Exercises

1. Construct an FSA for the following PL/I structure. Which partial name references are ambiguous? Which are illegal?

```
DECL 1  A,
     2  B,
        3  C BINARY,
        3  D  BINARY,
     2  C,
        3  D,
           4  E  BINARY;
```

2. Write a grammar for PL/I structures, at least the level and name parts. The primitive type specifications are complicated and may be omitted or generalized. Then assume that space for the structure is allocated in linear, contiguous memory. Develop an algorithm for computing the offset of any data object, and finally an algorithm for accessing any data object, given the necessary indices.

8.3.3.3. Pascal Structures

In Pascal, structures are created through the use of statements called *type* declarations. A type declaration specifies a data structure, but does not allocate memory space for it. There are several forms, but the one we are interested is

```
type T = record S1 : T1;
                S2 : T2;
                S3 : T3;
                 .
                 .
                 .
                Sn : Tn
         end
```

The S_i are called *selectors*. The T_i are either other type identifiers or data object declarations, e.g., a simple object or an array. An array of elements of type Z is written thus:

array [3..9] of Z

The data structure represented by the type T is shown in figure 8.6.

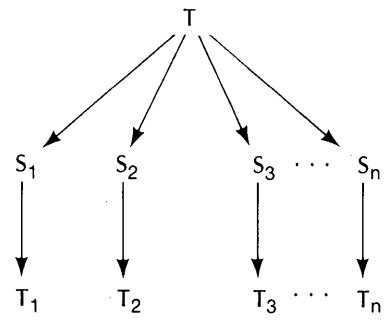


Figure 8.6. Compile-time data structure of a general Pascal RECORD declaration. T and T_i stand for type names, and the S_i are selector names.

Let us illustrate the Pascal convention with an example, and then we shall describe the kind of compile-time structures needed. This example defines a personnel data structure for Consolidated Widgets, Inc., or "CONSWIDGET" for short. The officers of the company enjoy their own section of the data structure, but share their VITA with all the other personnel. Note that it is necessary to define a given kind of data structure only once; it may be shared by any other structure as needed.

```
type NAMEPART = array[1..20] of char;

type NAME = array [1..3] of NAMEPART;

type STREETADDRESS = record STREETNUM : integer;
                            STREETNAME : NAME
                     end;

type VITA = record NAME : NAME;
                   AGE : integer;
                   SS : array[1..3] of integer;
                   SALARY : integer;
            end;

type OFFICER = record TITLE = (pres, vp, secy, treas,
                               boardmember, boardchairman);
                      VITAE : VITA
               end;

type ADDRESS = record STREET : STREETADDRESS;
                      POBOX : integer;
                      APT : integer;
                      CITY : NAME;
                      ZIP : integer
               end;

type COMPANY =
    record NAME : NAME; {the two NAMEs mean
                         different things—this sort of
                         name conflict is legal}
           HDQTRS : ADDRESS;
           OFFICERS : array[1..N] of OFFICER;
           PERSONNEL : array[1..M] of VITA
    end;
```

So far, no data space will be allocated for any of this structure. The following Pascal declarations create data spaces corresponding to the designated type:

```
var PERSONNELFILE : PERSONNEL;
    {creates a data object of personnel vitae}
var COMPANY : CONSWIDGET;
    {creates a data object of an entire company,
         Consolidated Widgets, Inc.}
var LEONARD : VITA;
    {creates a data object with name, age, social
        security number, etc.}
```

An example of a LEONARD data object is shown in figure 8.7. It consists of three NAMEPARTs, each an array of 20 characters, an integer AGE, an array of three integers SS and an integer SALARY.

It is clear that the "COMPANY" type structure, when expanded, is quite large, yet its description and its manipulation (as we shall see) can be very concise.

A reference to a Pascal data object is through a composite name, similar to that used in PL/I. For example, LEONARD's last name is accessed by

LEONARD.NAME[3][I]

where the I subscript indicates the specific character in the name. Pascal has

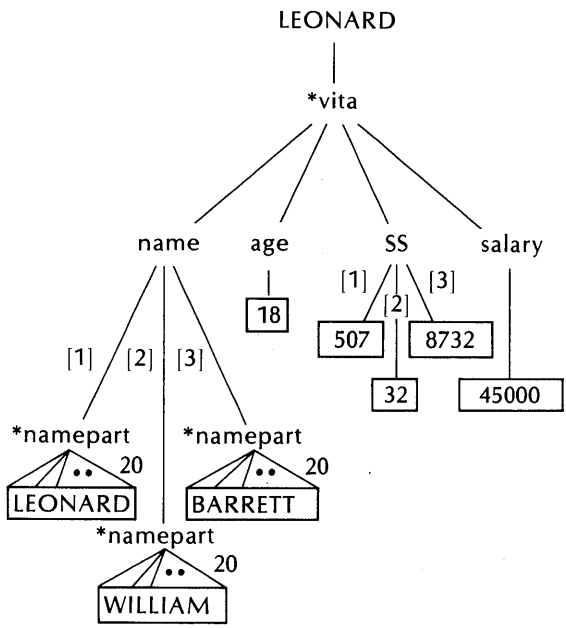


Figure 8.7. Example run-time data structure for a VITA structure.

no operations on a set of primitive data objects, as does PL/I and Cobol. Every name must refer to a primitive object.

The pair of subscripts [3][I] may also be written [3, I]. However, the subscripts may not be moved to the right; we shall see why not shortly.

The names may be abbreviated, but not in quite the same way that PL/I permits. Pascal provides a structured means of supplying partial names, the *with* statement. An example follows.

```
with LEONARD do
begin
   var I,J: integer;
   for I:=1 until 3 do
   begin
      J:=1;
      while NAME[I, J] <> ' ' do
      begin
         print(NAME[I, J]);
         J:=J+1
      end;
      print(' ');
   end
end
```

The two occurrences of NAME in this program are effectively preceded by "LEONARD." We shall see that this is more than just a convenience for the programmer, it can also reduce code and execution time if several operations on the elementary data objects appear in the "with" block.

The general form of a "with" block is:

```
with <name-list> do
begin
   ...
end;
```

```
<name-list> ::= <name-list> , <composite-name>
             ::= <composite-name>
```

The composite names in the <name-list> may contain indexing or whatever, including indices computed during execution. Thus in the CONSWIDGET data structure a legal "with" block is

```
with CONSWIDGET.OFFICERS[X+3]. VITAE do
begin
```

```
    NAME:= . . . ;
    SS := . . . ;
        .
        .
        .
end;
```

Pascal "with" blocks may also be nested, and an inner one may particularize a variable name heading started by an outer one, e.g.,

```
with CONSWIDGET do
begin
    .
    .
    .
    with OFFICERS[X+3] do
    begin
        .
        .
        .
        with VITAE do
        begin
            .
            .
            .
        end
    end
end
```

Pascal Type Symbol Table Structure. We now discuss the data structures internal to a Pascal (or similar) compiler that will support the "type" and "with" structures.

A starting point is the so-called "type structure," illustrated in figure 8.8. for the CONSWIDGET type given earlier. This is a directed acyclic graph (DAG) decorated with a set of attributes as shown. Each of the type names is marked with an asterisk (*); these names will not appear in a composite name. An elementary data object type is marked with a dagger (†). An array of objects or types is indicated by an "array" node, for example,

array [1..3]

The linkage in figure 8.8 clearly reflects this set of type declarations, and it can be constructed through productions and DAG-building semantics on the productions by methods discussed in chapter 7. We shall see that each of these links is necessary, including the link from the right-most child of a type node to its parent.

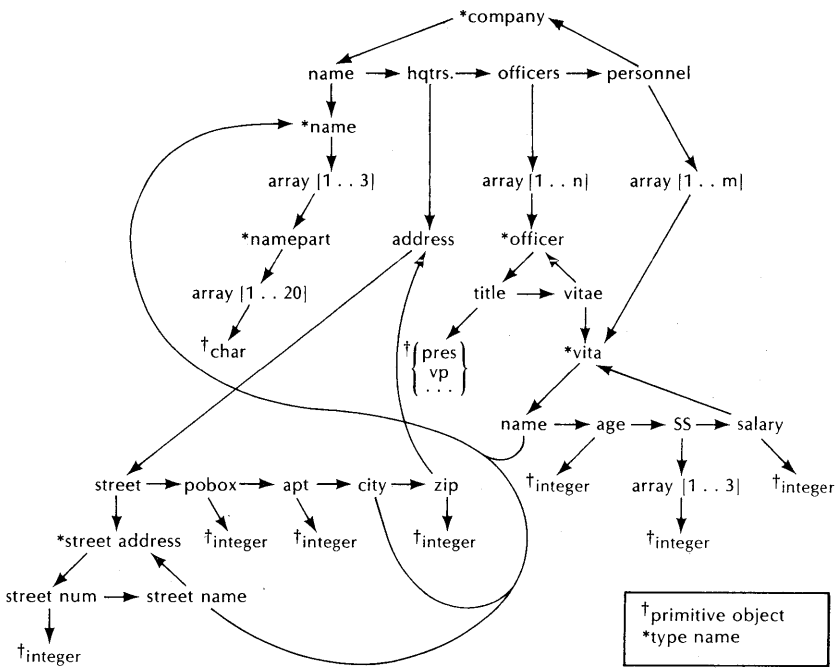


Figure 8.8. The compile-time structure for the COMPANY type structure.

The size of this structure is proportional to the number of statements in the type declarations; there is no expansion of data arrays, etc.

In addition to the links shown, there will also be a set of links from an access system to each of the simple names, to facilitate the rapid location of a given name.

We have several instances of identifiers with two or more uses, e.g., NAME. Pascal permits duplicate name use provided that no ambiguity exists. This feature causes some trouble in searching for names, especially in a "with" block, which we shall discuss later.

Inside a "with" block, a *with list* will be nonempty. Its form is shown in figure 8.9, simply a linear, linked list of nodes that indicate "type" nodes in the type structure.

The "with" list shown in figure 8.9 corresponds to the nested "with" blocks:

```
with CONSWIDGET do {creates the "*company" node}
begin
```

```
    .
    .
    .
    with HQTRS.STREET do  {creates the
                "*streetaddress" node}
    begin
          {with list has the form figure 8.9 here}
    end;
     {drop the "*streetaddress" node here}
end
  {empty the with list here}
```

Note that the new "with" list nodes are added at the top level and are removed last-in, first-out.

Now consider the problem of locating a composite name in the symbol table structure. Suppose first that a fully qualified name is given, e.g.

CONSWIDGET.HQTRS.STREET.STREETNAME[2, 5]

We need locate only CONSWIDGET through the name access system, making sure that we locate a data name, not some other kind. That entry should point into the type structure, and we simply follow through the structure by comparing selector names only. In the process, we note array dimensions. Eventually, this process must lead to an elementary object.

Suppose instead that we are inside the inner "with" block given above, covering the head CONSWIDGET.HQTRS.STREET, and we are confronted with the variable name

STREETNAME[2, 5]

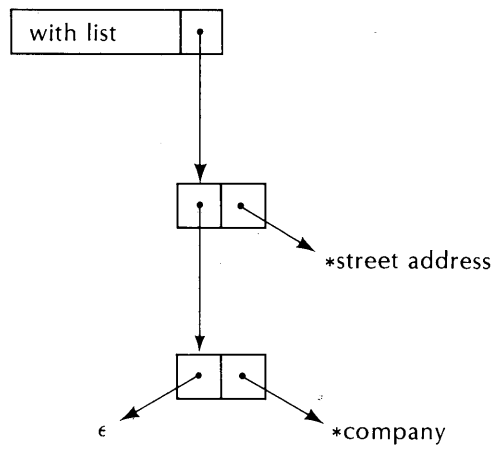


Figure 8.9. An example WITH list.

The name access system can locate a STREETNAME (maybe several, if we kept looking); we find that STREETNAME is not a data name, but a selector name. We know its location in some type structure, and as a selector, it must have a "type" parent. We locate the parent, using the reverse link from the right-most sibling to the parent, and the sibling links shown in figure 8.8. This process yields the type name "*STREETADDRESS". We then search the "with" list for the pointer value of "*STREETADDRESS" and of course find it in the topmost cell (see figure 8.9).

If we had instead the partially qualified name

OFFICERS[5].TITLE

then the second node in the "with" list would match the type pointer "*COMPANY".

Now Pascal permits name conflicts. We must therefore alter our strategy somewhat. If the above search of the "with" list fails to yield a match, then we may be looking at the wrong identifier in a name chain, i.e., we could be looking at a selector or type name that conflicts with some other valid name. We therefore must repeat the search process with the next matching name in a name chain, until we either find a sensible fit or the chain is exhausted.

With list construction. We construct the "with" list by adding a new node for every "with" list name. Now the "with" names are composite in general, but are incomplete. They may have a missing prefix (because we are in the scope of some covering "with" block) or a missing suffix (to be supplied by names in the block).

We therefore use the given strategy for each "with" list name, except that we work down into the type structure only as far as the name reaches. The last member of each name will be some selector; that selector must point to a type name, possibly through an array node, or else the name is in error. A pointer to that type name is then placed in the "with" list.

For example, consider the inner "with" name in the example:

```
with CONSWIDGET do    {creates the "*company" node}
begin
    .
    .
    .
    with HQTRS.STREET do  {creates the
            "*streetaddress" node}
    begin
        .
        .
        .
    end;
end
```

The name "HQTRS" is a selector name, whose parent is "*COMPANY". The "with" list contains one node for "*COMPANY", indicating that the front end of this name is covered. We may then continue down the type structure, ending on the STREET selector node; this node points to "*STREETADDRESS", which is what must be entered in the "with" list.

Exercises

1. Construct the type structure for the following set of Pascal types and variables:

```
type ALFA=array [1..10] of char;
     STATUS=set of (married, widowed, divorced, single);
     DATE=record MO:(jan, feb, mar, apr, may, jun, july,
                     aug, sept, oct, nov, dec);
                 DAY: 1..31
                 YEAR: integer
            end;
     PERSON=record
              NAME: record FIRST, LAST:alfa end;
              SS: integer;
              SEX: (male, female);
              BIRTH: DATE;
              DEPDTS: integer;
            end;
end;
```

2. Discuss implementation of a Pascal type structure and "with" list from productions. Could these be constructed in one stack?
3. Given the CONSWIDGET structure, show the "with" list at the inner block (→) of the following program segment:

```
with CONSWIDGET do
begin
   with OFFICERS[N], HQTRS.POBOX do
   begin
→
   end
end
```

Give examples of legal names at this inner level.

4. Are ambiguous Pascal names possible? Give some reasonable definitions, and develop an algorithm for checking for ambiguous name references. Note that partially qualified names are not permitted and that a primitive data object must always be specified; Pascal contains no operations on a set of primitive objects.

8.4. String Tables and Their Access

A *string table* is some storage system that provides for the efficient insertion and retrieval of identifiers (the strings.)

We shall present four rather different search methods. These have applications other than in compilers, for example, data base systems and sort systems. Of the four, the *hash access* method seems to be the method of choice for compilers since it is demonstrably the most efficient for the table sizes encountered in typical programming languages.

These access methods have certain properties in common, illustrated in figure 8.10 for the linear method. Three stacks are maintained, a *scope* stack that keeps track of the identifier domains and a *pointer* stack that carries links to the *identifier* stack.

Two identical names in different scopes may be entered twice or once in the identifier stack, depending on whether one is interested in minimum identifier stack size or minimum access time. In order to share name space, it is necessary to search the entire name table, not just the current scope, on entering a new name. It is also necessary to maintain an additional pointer to the identifier table in the scope stack, in order to reduce the identifier stack correctly at the end of a block.

Identical names in different scopes are rare, hence it is probably not worth the additional search time to attempt to share names. We shall assume that names are not shared in the remaining discussion.

Upon entering a new block, a pointer table top-of-stack (TOS) index is pushed in the scope table. This action effectively "marks" the name table stack.

Upon leaving a block, the TOS scope table index is used to reduce the pointer stack, and the resulting pointer stack TOS index is used to reduce the name table. If the names from this scope are needed later (in a multipass compiler), they may be transferred to secondary storage before reducing the stacks.

Upon a declaration, a search is made in the current scope (as defined by the pointer stack TOS and the scope TOS index) for the name. If the search succeeds, we either have additional attributes for an existing name or a multiply-declared name. If the search fails, the name is pushed in the identifier stack and a pointer to it pushed in the pointer stack.

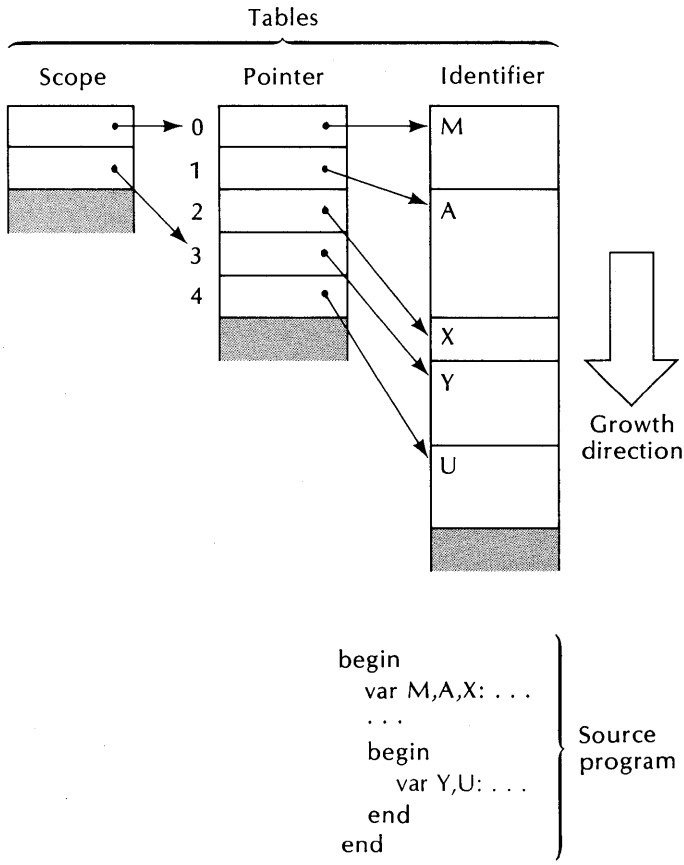


Figure 8.10. Linear name table-access method.

Upon a reference, a search is made in the entire table for the name, using the pointer stack indices. The search is conducted from the pointer stack TOS inward and stops on the first match or the bottom of the stack. If the name is not in the table, we may have an implicit declaration (as in Fortran) or an undeclared identifier. By stopping on the first match and by searching from TOS down, we locate that name most recently declared, hence the one declared in the innermost block.

The total declaration time t_D for a program will be different than the total reference time t_R for the following reasons:

1. The current scope is being built in the declarations, while it is essentially complete for references. (The truth of this assertion depends on the language. We are assuming Algol declaration rules for our discussion).

2. If the source program is free of declaration errors, an identifier will not be in the table for a declaration search, but will be in the table for a reference search.
3. A reference search may span the entire table, while a declaration search spans only the current scope. This search scope makes little difference in a language with only one block level, but makes considerable difference in a block-structured language. The difference is also influenced by the usage patterns of identifiers. The most heavily used identifiers are usually locals, and the next most heavily used identifiers are globals (block level 0).
4. A given identifier is declared once, but may be referenced several times in its block.

In order to make reasonably meaningful comparisons among the name access methods, we will develop "decl" and "ref" average times separately and make reasonable assumptions about the remaining factors. In section 8.4.5, the net access times of the four methods are compared. It is shown there that the hash access method is measurably superior to the others in access time. However, it requires somewhat more memory space than the others.

8.4.1. Linear Access

This method, the simplest of the four access methods, is illustrated in figure 8.10. On a declaration, a new name is simply pushed into the identifier table, and its index pushed in the pointer table. The names are clearly unordered, so that a search requires a comparison with each of the names in some region of the pointer table.

On a declaration, a search through the entire present scope is usually made. If the scope contains n names on the search, then the search time is

$$kn$$

where k is some unit comparison time. On each declaration, another name is added to the table, increasing n. If N′ names are declared, then the net declaration time for one block is

$$t_D = \sum_{i=1}^{N'} ki = kN'(N'-1)/2$$

The net reference time per block depends on the total number of active identifiers N in the table and on the usage pattern. Let us assume that references are distributed uniformly among the covering blocks and the current block. We know only that $N \geq N'$ and that each referenced identifier will be in the table (very nearly—barring programs with a large number of undeclared identifiers.) The average access time per identifier will be

$$kN/2$$

since we need scan only halfway through the table to find a given identifier (some will be near the top, some near the bottom of the identifier stack.)

We also need the average number of references per declaration b; then given N′ identifiers in the block, we have bN′ references total, for a net reference time per block of

$$t_R = kbNN'/2$$

The block entry and block exit times are both small and independent of the number of identifiers.

If a sorted symbol table listing is required at the end of a block, the sort may be applied to the current scope of the pointer stack. A "shell sort" requires $n(\log_2 n)$ operations to sort n items, hence a net time

$$kN'(\log_2 N')$$

is required for the sort.

8.4.2. Binary Access

Suppose we are in the kitchen and wish to eat a cake, but learn that there is a needle in it. The cake knife is in the kitchen and can't be moved, and a metal detector is in the garage and mustn't be brought to the kitchen. The detector says only whether or not metal is in a piece of cake; it doesn't have the resolution to locate it.

An optimal approach is to cut the cake in half and take one half to the detector. It indicates whether the needle is in this half or the other. We eat the half without the needle and cut the remaining half in half, etc. On each trip to the garage, we eat another half of the remaining cake and can eventually eat all of it—the successive divisions will reduce the cake down to the needle itself. If the cake conceptually consists of N pieces the size of a needle, then we need only $(\log_2 N)$ divisions to complete the process. This method is obviously a sizable improvement over a linear search, which would require cutting the cake into N pieces and taking each piece to the metal detector.

A binary search for a new identifier starts at the middle of some region of the pointer table (initially the center of the table.) A comparison with the center identifier then indicates which of the two halves should be considered for the next region. If N identifiers exist in the table, then $(\log_2 N)$ comparisons are needed to locate an identifier.

The binary access method is illustrated in figure 8.11. Each new identifier is pushed in the identifier stack as usual. Identifiers are not shared, hence a given name may appear more than once in the identifier stack.

The pointer stack contains indices to the identifier stack as usual; however, it is arranged so that at all times, the names are sorted as seen through the pointer stack. Duplicate names carry adjacent pointers. The scope stack

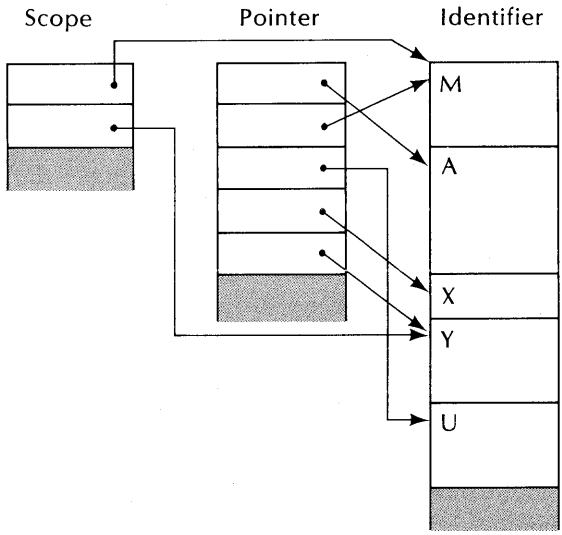


Figure 8.11. Binary name table-access method.

points to the identifier stack, not to the pointer stack; the latter does not have clearly delimited scope boundaries.

On a block entry, the current identifier TOS index is pushed in the scope table. The cost of this operation is small.

On a block exit, the pointer table must be purged of references to identifiers in the block being left, and then the pointer table must be compressed, which can be done efficiently in one scan of the pointer table. The scan moves pointers downward in the table, discarding those that belong in the current scope. Note that identifier comparisons (relatively expensive) are not involved in this operation.

A *declaration* requires a search followed by a sorted insertion. The insertion requires moving half the pointer table down, on the average, to make room for the new entry. Its location is clear from the search result. The move operation per item inserted is

$$a+k'n/2$$

where "a" is some move overhead time and k′ is the time per unit move. Many computers have a fast move instruction that can carry out this operation, so that "a" and k′ will be fairly small numbers.

The net time for a set of N declarations is then

$$\sum_{n=1}^{N} (a+k'n/2) = aN + k'N(N-1)/4$$

which must be added to the search and comparison times for the declarations, given by

$$\text{Search time} = k\left(\sum_{n=1}^{N} (\log_2(n)+1)\right) \sim kN(\log_2(N)+1)$$

On a reference, a search of the whole table must again be carried out. Now a reference match will be found before the search narrows down to one pointer. The net reference time is therefore approximately

$$t_R \sim kN'(\log_2(N'))/2$$

Although the reference time is proportional to the unit comparison time k, it increases much more slowly with N′ than for the linear search method and will be smaller, unless blocks are typically small and global references infrequent.

The search algorithm should examine the neighborhood of a "found" identifier for other identical identifiers and choose that one in the most local scope. This selection can be made systematic by arranging that the most recently entered identifier have the greatest index of a set of identical ones, rather than just any appropriate index. Then on the search, there must be an added search toward larger indices to find the most recently entered identifier.

A sorted symbol table report obviously requires no sort upon a block exit. However, the pointer table must be completely scanned, and only those identifiers within the current block selected through the use of the scope table and the identifier table indices.

8.4.3. Tree Access

Figure 8.12 illustrates the tree access method. On a declaration, a new identifier is pushed into the identifier stack as usual. Then a new tree node is created, consisting of a "left" and a "right" pointer pair, along with an identifier pointer element. The left and right tables constitute a binary tree description. Each node either has no children, a left child, a right child, or both; ϵ indicates no child. In placing a new identifier in the tree, we compare the new one with the existing ones, then walk down the tree starting at the root. We place the new node in the first available child position. At each node, we move left if the new identifier is lower in collation order than the node and to the right otherwise.

For example, consider a new identifier B. Then B $<$ M so we move left and

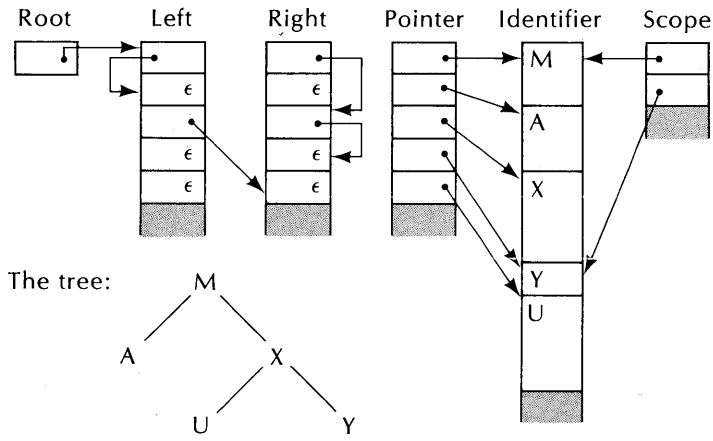


Figure 8.12. Binary tree name table-access method.

compare B with A. Now B > A, and A has no right child, so B goes into A's right child position. The new B entry has empty left and right children.

The average declaration time is $k(\log_2 N)$, plus a small time to add the four table entries. We do not have to move a list of pointers as we did with the binary method, so we appear to have a time improvement. However, k is somewhat larger for this time than for the binary search method, since a binary tree will never be completely balanced, which increases the average access time. In the binary search, the searching is always balanced.

The average use time is $k(\log_2 N)/2$; an identifier will be found halfway down the tree on the average.

The entry time is negligible; we merely add another item to the scope table.

The exit time is significant. We must walk through the tree and locate every node that points to an out-of-scope identifier (an identifier about to be dropped); this node must be changed to "empty." Then the identifier, left, right, and pointer stacks may be cut back.

A sorted symbol table is generated by sorting a portion of the pointer table. Alternatively, a left-to-right tree walk yields a sorted list—those identifiers not in the current scope are ignored.

As in the binary search method, some rule must be established to distinguish identical identifiers in different scopes, e.g. always placed to the left of an existing one. Then in the search, given a match, the comparisons should continue down the left side until a mismatch is found; the last matching one is that most recently entered.

8.4.4. Hash Access

The hash access method is illustrated in figure 8.13. A *hash function* is defined on the class of identifiers; this function maps every identifier into an integer between 1 and h, where h is a fixed hash table size. Thus in the figure, X maps to 2, M and A map to 4, and Y and U map to 6. It is not essential that the hash function map every identifier to a distinct integer (indeed this is impossible if the number of identifiers is greater than h), but it should provide a reasonably random and uniform mapping. We call the hash function value for some identifier its *hash code.*

Given the hash code of an identifier, we enter the hash table directly through the hash code as an index and search for the identifier along a chain; if the identifer is in the table, it must be along that chain. We enter each identifier at the head of its hash code chain. Note that a declaration search need go down a chain only until the present scope is left. For a reference search, a chain must be followed to its end, if necessary, since an identifier may be in any scope.

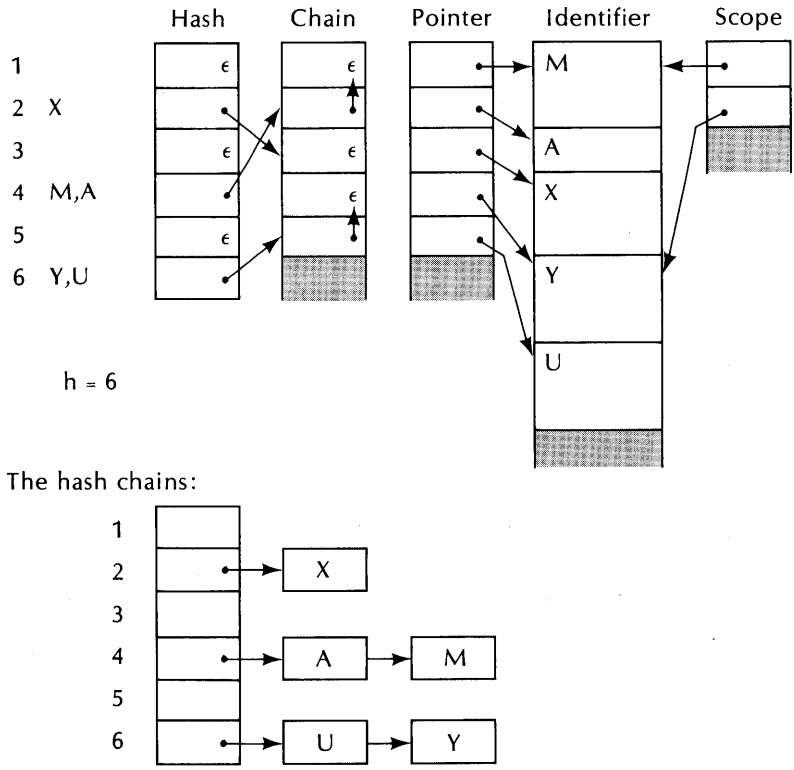


Figure 8.13. Hash name table-access method.

A *chain* table is shown in figure 8.13 to hold chain pointers; ϵ stands for the end of a chain.

For example, identifiers M and A hash to the same code, 4. Identifier M is entered first, by pushing it into the identifier stack. Corresponding entries are pushed in the pointer and chain stacks; the new chain index is 1, so the hash table entry is moved to the chain table (empty) and the hash table entry is set to 1. Eventually, the A is encountered. Repeating this process, we move the hash table entry to the new chain table entry (index 2) and place 2 in the hash table.

The configuration after all five identifiers have been entered is shown in the figure.

The declaration time for a name is kN/h. On the average, each hash chain contains N/h names from the current scope, and we must search one of them completely to establish that the name is missing.

The use time is kN′/(2h), since on the average we hit the identifier halfway down a chain. Note that these times are essentially linear search times, reduced by a factor of h, the number of hash buckets.

Some overhead is incurred in computing the hash function, but it can be kept small by using a simple hash algorithm.

The entry time is negligible. The exit time is appreciable. We must search down all chains of the hash table entries, and relink the chains to the previous scope names. However, since the names are encountered last-in first-out, only the present scope name chain entries are scanned. Those deeper in the chains can be ignored.

The block exit time can be reduced through one of the following schemes:

1. Push a copy of the hash table into the scope stack on block entry, in addition to the TOS indices of the chain and pointer and identifier stacks. Then on exit, the hash table may be restored without a chain search.
2. Maintain a hash table stack, and start a new hash table, initially empty, on block entry. On block exit, it may simply be discarded. However, a reference search must then work through each of the stacked hash tables. No special time penalty is paid for this search, however, since each of the chains is smaller by the number of hash tables, on the average, and each of the tables is easily entered through the common hash code.
3. Add a set of pointers through the table entries that constitutes a linked list of entries that must be at the top of the hash chains upon block exit. This is a rather expensive approach, since a pointer cell is needed for each hash table cell and entry.

The names are not sorted in this scheme, so a sort algorithm must be applied to the pointer table to list the symbols alphabetically.

Bounded Table Hash Access

Sometimes we can guarantee that the number of symbols entered are less than or equal to h. If so, we don't need the overflow table; we can simply apply a *rehash* algorithm repeatedly until we find an empty cell in the hash table. Morris [1968] has shown that this algorithm is superior in performance to a hash chain method.

This method is useful for tables with a fixed number of entries, e.g., a table of reserved words for a compiler or a table of assembler mnemonics. However, it is usually unnecessary to maintain two distinct name tables—the reserved names can be accessed by the same algorithm as user identifiers. Their reserved status can be inferred from their position in the table.

An optimal strategy for such a symbol table, from Morris, is expressed by the following Pascal program:

```
var C,R: integer;
var IDENT: array [1..n] of char;  {identifier}
var HASHTAB: array [0..h] of integer;  {hash table}

procedure RAND: integer;
begin
   R:=(R*5)mod(h*4);  {h is hash table size}
   return(R/4)
end;

{the hash access algorithm starts here}

C:=HASHIT(IDENT);    {returns a hash code}

while HASHTAB(C)<>NULL do
begin  {something in this hash chain}
   if NAMETAB(HASHTAB(C))=IDENT then return(TRUE);
        {found if match is found}
   C:=(C+RAND)mod(h);  {try another hash code}
end;
return(FALSE)  {not found}
```

The average number of probes required to locate an item known to be in the table is

$$E = 1 + (N/(h-N+1)) \sim 1/(1-a)$$

where h is the hash table size, N is the number of items in the table, and $a = N/h$ is called the *load factor*. Some sample values of E:

Load Factor a	E
0.1	1.11
0.5	2
0.75	4
0.90	10

A reasonable load factor is 0.75; 25 percent of the table is wasted, but an average of only four probes is needed to locate a name.

Hash Functions

We now consider the problem of a suitable hash function. We need a simple algorithm that will map a class of commonly used identifiers as uniformly as possible into an interval 1..h, where the hash table contains h buckets. Let us consider a simple function, and evaluate its hash behavior:

$$C := \text{FIRST} + \text{LAST} + \text{LENGTH} - m$$

where FIRST and LAST are the numeric forms of the first and last characters of the identifier and LENGTH is its length. The parameter "m" is a constant, defined as follows.

Suppose the characters are ASCII 7-bit coded and consist of 1 to 15 letters and digits, starting with a letter. The ASCII codes are:

"A"=65, "B"=66, . . . "Z"=90, "0"=48, "1"=49, . . . "9"=57.

Now let S=FIRST+LAST+LENGTH. The least S is 115, for the identifier "A0" (A single letter identifier would have a larger S, since the letter is added in twice). The largest S is 195, for the identifier "ZXXXXXXXXXXXXXZ" (the X's may be anything). We therefore set m=114. The largest useful h is then (195−114)=81. The class of identifiers can clearly fill the hash code interval 1..81, so that no hash table cells will go completely unused.

An experiment with a program containing 623 identifiers and a hash table with 81 buckets yielded the following summary statistics:

Chain Length L	Number of Chains with Length L
0	24
0/4	41
5/9	9
10/14	9
15/19	12
20/24	6
25/27	3
>27	0

The net cost of accessing the 623 identifiers through the hash chain system was 9417 name comparisons. Now the average chain length is 7.7 names (623/81), so that the 5 to 9 chain length range should predominate. Instead, we find that almost half the chains carry less than 5 names, and several chains carry more than 25 names. Clearly, something is wrong with this hash function.

We clearly cannot leave an identifier hash function to chance. We may end up with many of the cells unused or a group of cells much more heavily used than others.

Knuth [1973] showed that an effective hash function on some key K is simply a modulo h division or multiplication. The key K must be some integer, possibly formed from a few characters in the identifier by concatenation. Then

$$h(K) = K \bmod M$$

where M should be a prime number yields a reasonably random hash code. Values of M which divide $r^k + a$, or $r^k - a$, where k and a are small numbers and r is the radix of the alphabetic character set (usually 64, 256, or 100, depending on the machine), should be avoided.

Suppose we have 32-bit integer arithmetic. Then we can form K from the identifier string $a_0\ a_1 \ldots a_n$ by the following algorithm:

```
var D: double integer;

if n>4 then D:=[a0, a1, an-1, an]
        {formed by concatenating the
         first two and the last two characters}
else D:=[blank, blank, . . ., a0, . . ., an]
        {left-most blank fill}
```

The following table shows the results of an experiment with this same program, containing 623 identifiers but using a Knuth hash function:

Chain Length L	Number of Chains with Length L
0	0
0/4	9
5/9	54
10/14	17
15/19	1
>19	0

Clearly, the chain lengths are more uniform. The net access cost is 5,794, about half that for the hash function above. The dominant chain length is closer to the expected average, 7.7, and there are no unused or unusually large chains.

8.4.5. Comparison of Access Methods

The four access methods described may be compared through their required space or time. In comparing space, we note that all four require the same identifier, scope, and pointer space. We therefore compare the space required in addition to this:

- Linear: 0
- Binary: 0
- Tree: 2*N, where N = number of table symbols
- Hash: h + N

A comparison of access times is somewhat more difficult. We have previously developed formulas for entry, exit, declaration and reference (use) times. These formulas are summarized in the next table.

Summary of Access Times

	Entry	Exit	Declaration	Use
Linear	10	10	25*N'*(N'−1)	25*N*N
Binary	10	16*N	25*N*(log(N)−1)	25*N*log(N) +5*N+k'*N*(N−1)/2
Tree	10	8*N	35*N*(log(N)−1)	35*N*log(N)
Hash	10	8*N'	25*N'*(N'−1)/h	25*N*N/h

Each of the table entries represents some net access time, in arbitrary units, for a typical program block. Here N is the number of identifier declarations in the block, and N' the number of identifier references. Parameter h is the number of hash buckets. The remaining factors have all been estimated, based on our experience. For example, the use time for tree access is similar to that for a binary sort, except that the factor (35) is larger because of tree imbalance. The binary sort exit time is larger than a tree exit time on account of the pointer table compression required.

We next fold these time estimates into the following set of frequencies, given by McKeeman [1974b]. These apparently are based on experience with a number of XPL (block-structured) programs:

f1, entry	f2, exit	f3, decl	f4, use
10	10	100	700

We then may compute an average unit access time, through the following averaging formula:

$$\text{Time} = (f1*\text{entry} + f2*\text{exit} + f3*\text{decl} + f4*\text{use})/(f1+f2+f3+f4)$$

We assume that $N' = N/5$, i.e., for every identifier declared in the current scope, there are four others in covering scopes in the symbol table. (This number 5 may be too large for an average program, but it doesn't affect our conclusions much.)

We also assume that the number of hash buckets h is 100.

The graph in figure 8.14 shows the net time as a function of N in the range 1 through 360. The hash table must be nearly full at $N=360$, yet the hash method remains superior to the others far beyond this N. In fact, the hash and binary methods have about the same time at $N=1200$: binary is 288, hash is

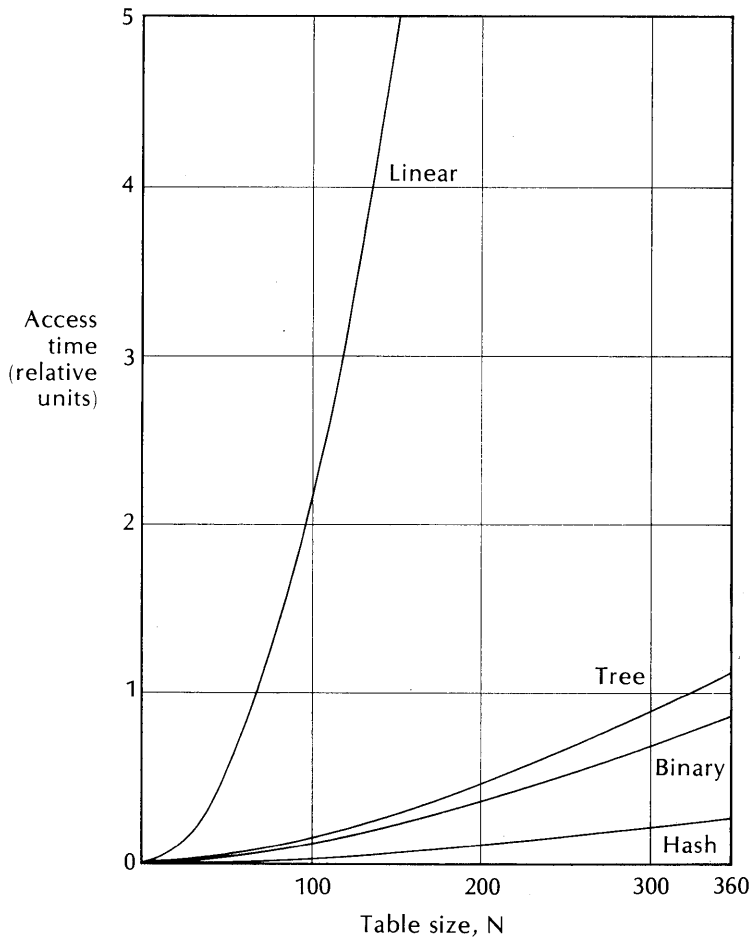


Figure 8.14. Comparative access times for the four access methods, linear, tree, binary and hash, as a function of table size.

279. At N = 1200, the average hash chain is 12 symbols long, requiring about 12 comparisons. However, 1200 is approximately 2^{10}, so the binary method also requires about 11 comparisons.

For N > 1200, the binary method is the most efficient; however, the efficiency of the hash method increases with the number of hash buckets h, so that a larger h might be justified for such a large string table.

The tree method is consistently poorer than the binary method; note that it also requires more storage. The linear access is orders of magnitude more costly in time than any of the other three. At only 100 symbols, linear access requires 100 times the search time as hash access. (This factor of 100 is the number of hash buckets.)

Summary of access methods

The most efficient access method, by time comparison, is the hash access method. It requires a function that maps an identifier into a finite range of integers 1 to h in a uniform manner. The hash code resulting from the mapping is then used to isolate the identifier to one of h separate linear chains of identifiers. The desired identifier is found by a linear search along that chain. Although the binary and tree search methods result in a sorted table, convenient for a symbol table listing, this apparent advantage is outweighed by the larger overhead in declaration and reference times.

8.5. Bibliographical Notes

The use of symbol tables in computer systems appears to have ample precedent. access methods, with detailed examples, are given by McKeeman [1974b]. Knuth [1968], vol. 1, chapter 2, contains a treatment of symbol table tree structures suitable for PL/I or Cobol names. Knuth [1973] discusses searching and hashing at length.

A formal symbol table and attribute system of considerable power and generality, called *property grammars* have been recently defined by Stearns and Lewis (Stearns [1969]). A good review of property grammars, with examples, is given in Aho [1972a], vol. 2, chapter 10; Aho also develops statistical formulas for the evaluation of the efficiency of the hash table search method.

CHAPTER 9

RUN-TIME MACHINE STRUCTURES

9.1. Introduction

In the last chapter, we developed symbol table structures that the compiler should maintain as a means of keeping track of (1) the attributes of the various data objects that will appear in code at run-time and (2) information related to the blocks and procedures of the source program. We now need to describe run-time structures which entails the allocation of variable space and mechanisms for dealing with procedure calls, parameter passing, and variable access.

We can develop a system of run-time structures without resorting to coding details of any particular machine by developing a minimal set of instructions for a fictitious machine that supports the data structure and procedure mechanisms required for Algol 60, Pascal, and PL/I. Embedded in these instructions is a subset that can support Fortran or Basic. Given an Algol-like language, those machine features that suit that language can be selected from this machine and perhaps simplified or modified to suit the desired language properties.

In each of the above languages the dimensions of vector or matrix data objects may be dynamically altered, though in rather different ways in the different languages.

A more general dynamic data allocation system is required in languages such as APL and Lisp, where data may be allocated and restructured "on the fly" as it appears in various executable expressions. Data objects are never declared as such in APL; their characteristics are inferred from operations that set their values. We shall not discuss data allocation for APL and Lisp, although certain of the ideas expressed in this chapter are applicable to interpreters of these languages.

The fictitious machine instructions developed in this chapter are sufficiently well defined that a simulation program can be written in any common programming language. We suggest that such a simulator be written and used to study the machine operations developed in this chapter.

9.2. Run-Time Structures for Algol-Like Languages

We now develop a run-time stack machine system that can support essentially all of the features of Algol 60, Pascal, and PL/I.

This machine, called the AOC machine (for Algol Object Code), we owe to Randell and Russell (Randell [1964]), with revisions by James Morris and the authors. Its mechanism focuses on the important problems of variable scope conventions, nested procedure declarations, allocation and assignment of memory space to variables, access of variables, indexed variable conventions, and procedure call conventions.

The AOC machine has a linear sequential data memory with locations addressed by h, h+1, . . . , −1, 0, 1, . . . , s−1, s, each of which can hold a numeric value (an integer or a real) or a truth value (TRUE or FALSE). The portion of data memory accessed by negative locations is called the *heap*, and the remaining portion is called the *stack*. The data memory is regarded as an infinite two-way tape; the locations may extend arbitrarily far in either the positive or negative direction during execution of a program. The largest positive location in use during execution is denoted by s and the most negative location by h. Both the heap and the stack are initially empty, hence initially h=0 and s= −1.

The AOC machine also contains a finite linear sequential program memory with locations addressed by 0, 1, 2, . . . , n. The contents of this memory and its size cannot be changed during execution. The program memory is initially loaded with program instructions and thereupon remains fixed until the end of the program. Location 0 is the first executable instruction. No data is located in the program memory. Although it could be used to hold constant data, difficulties in accessing such data through an address label would thereby be created.

We shall not be concerned about the number of bits per word, or possible packing of multiple elements per word. We shall also not be concerned about allocation and assignment of physical memory to the data and program memories. We assume that sufficient memory space is always available to support whatever program or operations are required of the AOC machine.

The contents of cell j in data memory is denoted by C(j). The notation

$$C(j) \leftarrow x$$

means "store the value x in location j." The notation

$$C(j) \leftarrow a,b,c$$

is equivalent to the set of commands

$$C(j) \leftarrow a;\ j \leftarrow j+1;$$

$$C(j) \leftarrow b;\ j \leftarrow j+1;$$

$$C(j) \leftarrow c;\ j \leftarrow j+1;$$

i.e., the set a,b,c are stored in consecutive locations in memory. Then the notation

$$x,y,z \leftarrow C(j)$$

is equivalent to

$$z \leftarrow C(j);\ j \leftarrow j-1;$$

$$y \leftarrow C(j);\ j \leftarrow j-1;$$

$$x \leftarrow C(j);\ j \leftarrow j-1;$$

so that when the list (a,b,c) is stored and later restored in (x,y,z), x corresponds to a, y to b, etc.

The contents of cell i in program memory is denoted PB(i). Since program memory is read-only, only operations of the form

$$x \leftarrow PB(i)$$

are legal. In fact, the function PB(i) will only be required in the master execution procedure, described later.

If $i<0$ or $i>n$ then PB(i) = "HALT". The master execution program may access PB(n+1), but no instruction outside the range $0 \leq i \leq n$ should ever be executed in a valid program.

The program memory is fixed at compile time, in principle at least. An implementation may defer some operations on the program memory to just prior to execution of the program. However, no alterations in program memory are permitted during execution.

There is also another small independent memory called the *display*. Its cells are numbered 0,1,2,...,b and always holds certain addresses of the data memory. Here, "b" is the largest display index, and will vary during execution. The display is designated D, e.g.,

$$D(3) \leftarrow 7$$

$$C(5) \leftarrow D(6) + 2$$

are possible operations. The display will track the current depth of nesting in nested procedures or blocks, so that the maximum b is the largest depth of nesting found in a source program. Wirth found that $b_{MAX} = 3$ was sufficient for a Pascal compiler (Wirth [1971a]), so that the display size can be quite small indeed. We shall also see that a copy of the display must be kept in the data memory, and is always accessible, so that as it turns out, the display is needed only for the sake of efficiency and clarity.

The machine also has four special registers for the variables i, s, h, and b.

The overall operation of the AOC machine is described by the master execution program below. Initially, data memory is empty, $b = s = -1$, and $h = i = 0$. The program memory is loaded with instructions, and then

execution of the instructions begins at location 0. The program will expand and contract the data memory and display through its instructions and will stop upon encountering the HALT instruction.

```
{AOC machine master execution program}
var INS: integer;
LOAD(program);
s:=-1;
h:=0;
b:=-1;
i:=0;
INS:=PB(i);    {first instruction}
while INS ≠ 'HALT' do
begin
   i:=i+1;
   EXECUTE(INS);    {execute the instruction}
         {Note: Execution of INS may change i}
   INS:=PB(i)       {fetch the next instruction}
end;
```

The description of the individual instructions and their rationale constitute the rest of this chapter. The memory and register structures of the AOC machine are shown in figure 9.1.

A complete list of the instructions and their effect is given in section 9.9; they represent an instruction set for full Algol support. We shall evolve toward that set by starting with fairly simple definitions, then expanding them as the need arises.

We shall see that these are powerful instructions, despite their apparent simplicity. They are also easily implemented in any of several different ways, namely:

1. Build a machine according to the specification of these notes, perhaps through a combination of hard wired logic and microprogramming.
2. Write an interpreter for the instructions. This interpreter might be a program that executes Snobol instructions on a CDC 6400, or an assembly language interpreter implemented on a microprocessor.
3. Write a program to translate AOC into machine code for some machine.
4. Write a set of macros that generate machine code in assembly language.

9.2.1. Arithmetic and Logical Expressions

An expression like

$$A + B/7 - 3*D$$

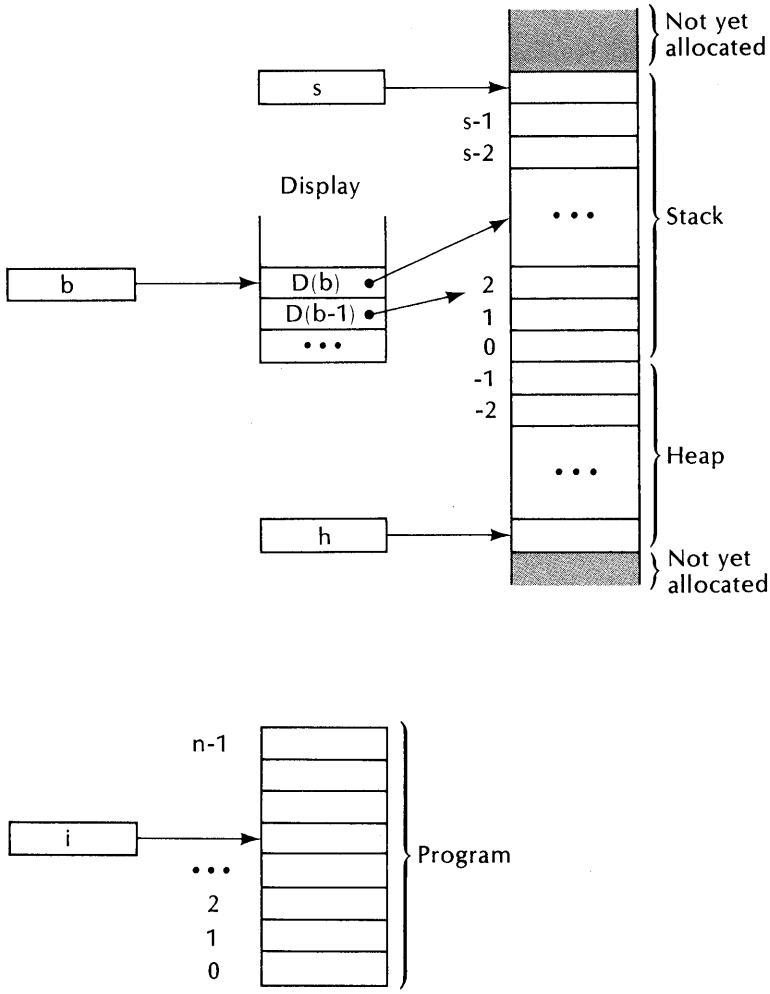


Figure 9.1. Memory and registers of the Algol object code (AOC) machine.

can easily be evaluated by a stack machine, as we have seen in chapter 7. A bottom-up or top-down compiler can simply emit stack load instructions on each variable and appropriate stack operations on each operator, as they are encountered in the parsing process. We are thus led to the following AOC arithmetic instructions:

```
LC n:  Load constant n
       s ← s+1
       C(s) ← n   {s marks the stack top}
```

LV a: Load value contained in address a
$s \leftarrow s+1$
$C(s) \leftarrow C(a)$

(*Note:* For the moment, an address is simply an integer. Eventually, an address will become more complicated.)

ADD: Arithmetic addition
$C(s-1) \leftarrow C(s-1) + C(s)$
$s \leftarrow s-1$

SUB: Arithmetic subtraction
$C(s-1) \leftarrow C(s-1) - C(s)$
$s \leftarrow s-1$

MULT: Arithmetic multiplication
$C(s-1) \leftarrow C(s-1) * C(s)$
$s \leftarrow s-1$

DIV: Arithmetic division
$C(s-1) \leftarrow C(s-1) / C(s)$
$s \leftarrow s-1$

Assume that A, B, and D in the expression $A+B/7-3*D$ are stored in locations -102, 120, and 200, respectively, and that $s = 300$. Then the following instruction sequence will compute the expression's value, leave it in location 301 and leave $s=301$:

```
LV      -102 {A}
LV      120  {B}
LC      7
DIV
ADD
LC      3
LV      200  {D}
MULT
SUB
```

The following instructions provide more arithmetic and logical capability:

NEG: Negate
$C(s) \leftarrow -C(s)$

NOT: Logical complement
(defined only for C(s)=TRUE or FALSE)
$C(s) \leftarrow$ if C(s)=TRUE then FALSE
else TRUE

EQ: Equal
$C(s-1) \leftarrow$ if $C(s-1)=C(s)$ then TRUE
else FALSE
$s \leftarrow s-1$

EQ compares the top two stack values as integers and replaces them by a single Boolean value.

LS: Less then
$C(s-1) \leftarrow$ if $C(s-1) < C(s)$ then TRUE
else FALSE
$s \leftarrow s-1$

AND: Logical and
$C(s-1) \leftarrow$ if $C(s-1)=$TRUE and $C(s)=$TRUE
then TRUE else FALSE
$s \leftarrow s-1$

The AND operation compares the top two stack values as Boolean variables and replaces them with a TRUE or FALSE, according to the usual meaning of logical AND. The operations AND and OR are only defined if the top two stack elements are in the set {TRUE, FALSE}.

OR: Logical or
$C(s-1) \leftarrow$ if $C(s-1)=$TRUE or $C(s)=$TRUE
then TRUE else FALSE
$s \leftarrow s-1$

9.2.2. Assignment Statements

The statement

$$A := B+1$$

is supposed to calculate the value of $B+1$, then set the cell assigned to A with the result. We introduce the instruction STD to accomplish this end:

STD a: Store direct in a
$C(a) \leftarrow C(s)$
$s \leftarrow s-1$

Then the statement $A:=B+1$ is executed by the instruction sequence

```
LV   B
LC   1
ADD
STD  A
```

We also must support access of indexed variables, e.g., A[I], where the address of A[I] is (address of A)+C(I). Recall that instructions may not be changed, so we cannot just change a STD A during execution of the program. We need instead a set of instructions LA, ST, and CONT as follows:

```
LA  a:  Load Address a
        s ← s+1
        C(s) ← a
```

This appears to be just like LC. However, as we add language features, addresses will get more complicated and LA and LC will differ.

```
ST:  Store
     C(C(s−1)) ← C(s)
     s ← s−2
```

ST expects an address in C(s−1). It stores C(s) into that address.

```
CONT: indirection
      C(s) ← C(C(s))
```

CONT expects an address on the stack top and replaces it with the value in that address.

An assignment statement of the form

$$A[I] := B[J]+1$$

can then be coded as:

```
LA    A   {Load address of A, for the sake of
            the ST later}
LV    I
ADD       {address of A[I] now on stack}
LA    B
LV    J
ADD       {address of B[J] now on stack}
CONT      {value of B[J] now on stack}
LC    1
ADD       {B[J]+1 value is in C(s),
            address of A[I] is in C(s−1)}
ST
```

Exercise

Write AOC code for the following statements:

```
A:=B*C - D*E + 75
B:=F AND (D OR E)
X[I+15]:=-Y/X[I-5]
```

9.2.3. Conditionals

There are several kinds of conditional constructs in the Algol-like languages:

1. Conditional expressions such as

```
if A=0 then 1 else 1/A
```

2. "One-armed" conditional statements such as

```
if B then B := FALSE;
```

3. "Two-armed" conditional statements such as

```
if B then A := 1 else D := 1;
```

4. Loop conditional statements such as

```
while A < 15 do A := A+1;
```

These and other conditionals may be handled by the following two instructions:

```
JP   l:  Jump to location l
         i ← l
```

Recall that i is the instruction register; it contains the location of the next instruction to be executed.

```
JIF  l:  Jump to l if top of stack is FALSE
         if C(s) = FALSE then i ← l
         s ← s-1
```

Note that we "fall into" the instruction following the JIF if the top of stack contains TRUE. In any case, the Boolean value on the stack top is removed.

Let us now translate the Algol control statements given above into AOC code sequences.

1. The expression

```
if A=0 then 1 else 1/A
```

is translated as:

```
      LV   A
      LC   0
      EQ        {TRUE or FALSE is left in C(s)}
      JIF  L1
      LC   1    {The THEN part}
      JP   L2
L1    LC   1
      LV   A    {The ELSE part}
      DIV
L2    . . .
```

We use symbolic names L1 and L2 to denote locations to simplify the discussion. When the program is loaded, of course, the L1 in the "JIF L1" must be set to the location of the "LC 1" instruction when it is found later on.

2. The statement

if B then B:=FALSE

is translated as:

```
      LV   B
      JIF  L1
      LC   FALSE
      STD  B
L1    . . .
```

3. The statement

if B then A:=1 else D:=1

becomes:

```
      LV   B
      JIF  L1
      LC   1
      STD  A
      JP   L2
L1    LC   1
      STD  D
L2    . . .
```

4. Finally, the loop statement

while A<15 do A:=A+1

becomes

```
L1  LV   A
    LC   15
    SUB
    NOT
    JIF  L2
    LA   A
    LV   A
    LC   1
    ADD
    ST
    JP   L1
L2  . . .
```

Of course, we can also translate simple GOTO statements using the JP instruction. For example, the nonterminating Algol program

```
M:  A:=A+1;
    goto M;
```

may be translated to

```
L   LA   A
    LV   A
    LC   1
    ADD
    ST
    JP   L
```

9.3. Stack and Heap Allocation

Most of the Algol-like languages provide for dynamic allocation of memory. The allocation can occur in one of two ways as follows:

1. Space is allocated upon entering a block and encountering a declaration. The upper and lower limits of an array declaration may depend on run-time variables. The allocated space may not be changed within the scope of the block and vanishes upon leaving the block.
2. Space is allocated upon executing some run-time function or is the result of a declaration (with new dimensions) of an existing OWN array.

Space for arrays of type (1) can be allocated from the stack. Space for arrays of type (2) require the heap and a memory manager, as we shall explain in greater detail later.

For now, we introduce three instructions that facilitate the dynamic allocation of stack and heap space:

```
INCS:  Increment stack pointer  (C(s) ≥ 0)
       C(s+C(s)) ← s;
       s ← s+C(s)
```

This expects a positive count N on the stack top. It allocates N words, and leaves the address of the allocated space on the stack top. The address may then be deposited somewhere, to form an array base address.

```
INCS n:  Increment stack pointer (n ≥ 0)
         s ← s+n+1
```

This form of INCS simply allocates a fixed number of words, n. No address is left on the stack top. Since n is known to the compiler, the addressing operations related to this INCS are compiler operations. The two forms of INCS could be combined through a rule that if n=0 in INCS n, the stack count is the top value on the stack.

```
DECH:  Decrement heap pointer
       h ← h − C(s)
       C(s) ← h
```

This modifies the heap memory extent (in either direction) by the number of words found on the stack top. The new h is returned and replaces the count word. This would be the address of an allocated heap array if C(s) > 0 initially.

```
DECH n:  Decrement heap by n words
         h ← h−n
         s ← s+1
         C(s) ← h
```

This decrements the heap pointer by n words, and pushes the current heap bottom on the stack.

```
DECS:  Decrement stack (C(s) ≥ 0)
       s ← s − C(s) − 1
```

This decrements the stack pointer by the number of words left on the stack top.

```
DECS n:  Decrement stack (n ≥ 0)
         s ← s−n
```

This decrements the stack pointer by n words.

9.4. Input-Output

The Algol 60 report contains no I/O definitions. However, we may easily add two primitive I/O operations. These are similar to those defined in Pascal:

- READ: A parameterless procedure which returns an integer read from some input tape as its value and also causes the tape read head to advance one position.
- PRINT: The value on the top of the stack is printed on an output tape or other medium, then removed from the stack top.

We conceive of input as a sequence of numbers on some input tape, and output as an endless tape, written one number at a time. Very little imagination is required to change the printing and reading mechanisms to accept characters, which, after all, can be considered special number codes.

Then

PRINT(READ)

prints the next input number on the output medium.

The AOC instructions for I/O are

```
READ:   Read number from input
        s ← s+1
        C(s) ← the next number on the tape
        (Tape input pointer moved one position.)
PRINT:  Print top of stack onto tape
        Print C(s) on the output medium
        s ← s−1
        (Tape pointer moved one position.)
```

We need one last instruction (for now), HALT, which simply stops the computer.

9.5. Blocks and Storage Allocation

We have not yet discussed allocation of variables to the memory. The easiest plan is simply to allocate variables in stack addresses 0, 1, 2, ...

This simple plan effectively gives us a Fortran machine, since in Fortran, each variable is given a fixed, permanent location in memory. However, in Algol, it is possible to declare some variables in the middle of a program, use them for awhile, then let them "go away"; such variables are said to have a limited scope (the region between their declaration and "letting them go"), and may not be accessible outside their scope. The advantage of the Algol system is two-fold: memory may be reused for different local variables, and the same name may be used in different scopes without conflict.

For example, the following Algol program reads three numbers and prints the first two in the same order as read, or the reverse order, depending on the sign of the third number.

```
begin integer A, B;  {start of scope 1}
      A := READ;
      B := READ;
      if READ < 0 then
      begin integer T; {start of scope 2}
            T := A;
            A := B;
            B := T;
      end;   {end of scope 2}
      PRINT(A);
      PRINT(B);
end     {end of scope 1}
```

The variable T is valid only within the scope 2; variables A and B are valid anywhere within scope 1. Variable T is only needed as a temporary for the interchange of A and B.

The range of use of variables is governed by the Algol *renaming rule*:

Algol Renaming Rule (TRR). If all the occurrences of an identifier declared at the beginning of a block are replaced by a new identifier which does not occur anywhere within the block, the meaning of the program does not change.

Note that the occurrence of the identifier in the declaration must be changed as well. For example, the program

```
begin integer X;
      X := 1;
      begin integer X;
            X := 2;
      end;
      PRINT(X);
end
```

is equivalent, by TRR, to

```
begin integer X;
      X := 1;
      begin integer Y;
            Y := 2;
      end;
      PRINT(X);
end
```

The point of TRR is that it leaves no doubt that this program prints "1", not "2". Another way of looking at TRR is that variable X in the first program statement is "suspended" by the inner block declaration of X; its value is picked up again at the end of the inner block. TRR clearly demands two different cells to be assigned to the two variables X and Y in the second program statement.

Now let us sketch a Fortran-style allocation system. To translate a program:

1. Use TRR so that every identifier has exactly one declaration. Let there be m different identifiers after this is done.
2. Translate the program (putting the first instruction in location 0) using the identifiers for addresses. Let the program translate into n AOC instructions.
3. Associate the m identifiers with locations 0, 1, 2, . . . , m − 1 in data memory. Then replace the identifiers in the instructions with their associated locations.
4. The first program instruction will be:

   INCS m

   to allocate m words for the variables.

The translation of the reversal program (above) by this procedure would then be:

```
0  INCS 3   {space for the three variables}
1  LA   0   {variable A}
2  READ
3  ST
4  LA   1   {variable B}
5  READ
6  ST
7  READ
```

```
 8   LC   0
 9   LS
10   JIF  20
11   LA   2    {variable T}
12   LV   0    {variable A}
13   ST
14   LA   0    {variable A}
15   LV   1    {variable B}
16   ST
17   LA   1
18   LV   2
19   ST
20   LV   0
21   PRINT
22   LV   1
23   PRINT
24   HALT
```

If we wanted to conserve data space, we could permit some of the variables to share the same space. Two variables can share the same data memory space if they are never simultaneously active and are both local. (Fortran variables cannot be shared.) Thus consider

```
begin integer x;
        .
        .
        .
      begin integer a,b,c;
              .
              .
              .
      end;
        .
        .
        .
      begin integer d,e,f,g;
              .
              .
              .
      end;
        .
        .
        .
end
```

The two sets of variables {a,b,c} and {d,e,f,g} are never active at the same time. They are also undefined upon entry to a block, so three of them could share the same storage locations. Then the above program would require only five storage locations:

- One for x in the outer block.
- Four for {x,a,b,c} in the first inner block.
- Five for {x,d,e,f,g} in the second inner block.

This Fortran-style scheme breaks down if recursive procedures are permitted in the language.

Exercises

1. Translate the following Algol program segment into an AOC instruction sequence:

```
integer I, J, K, L, M;
J:=15;
I:=0;
M:=3;
while I<READ do
begin
  M:=M+1;
  J:=J+I;
  if J>20 then J:=0
end
```

2. Write an AOC code sequence that will allocate space for a one-dimensional array (size n words) and fill it with zeroes. Consider two cases: (a) where n is a compile-time constant, and (b) where n is obtained by first evaluating some expression at run-time.

3. Suppose that several one-dimensional arrays are wanted, each of whose dimensions are evaluated during execution upon entering some block. How would they be allocated in the stack? Is it possible to write access instructions for them in AOC code, as developed so far? How might the array elements be accessed? Recall that the instructions cannot be altered at run-time.

4. Write an AOC program that reads a list of numbers terminated by -1, then counts and reports the number of 0's, 1's, 2's, ... , n's found. (Here n is the first number in the list, and the list contains only nonnegative integers, except for the terminating -1.)

5. Sketch an algorithm that allocates data space for variables and fixed-dimension arrays for a block-structured Algol 60 program, with maximum data space sharing.
6. Develop AOC code sequences to support a REPEAT UNTIL structure, as in Pascal.
7. Develop AOC code sequences to support a Pascal labeled CASE statement. Are any new instructions needed? If so, define them.

9.6. Procedures and Recursion

One or more procedures in Algol may be declared at the beginning of any block. A procedure declaration consists of two parts, a *heading* and a *body*. Its heading will be (for now) simply the key word PROCEDURE followed by its name and a semicolon. Its body may be any statement and most often is a block. Like other variables, a procedure name is known and may be called anywhere within the block in which it is declared; it is unknown outside that block. It is also known within its own block, hence may call itself.

Procedure semantics are governed by *the procedure copy rule*:

Procedure Copy Rule (TPCR). The effect of a procedure call shall be as if the body of the procedure were to replace the calling statement, then be executed. (Some complications will be brought out later.)

We may regard the copying as being done while the program is being executed by a machine which executes Algol programs directly, with no translation. For this simple case, the copying could be done at any time, even before the program starts.

Consider the following program, which prints the numbers on its input tape in reverse order:

```
begin procedure REV;    {procedure body beginning}
      begin integer X;
            X := READ;
            if ~(X=0) then REV;
            PRINT(X)
      end;     {end of the procedure body}

      REV;     {The procedure call}
end
```

Let us trace the execution of this program in a symbolic way. We shall do so by simply writing out the program steps that have been executed, and placing an arrow before the statement that is about to be executed. A variable may be defined, indicated: "X=6", or may be undefined, indicated: "X=?".

The first executable statement in the reverse program above is the call REV. Let the input tape contain the numbers 5, 6, 0. Then shortly after the first call, we have

```
begin procedure REV;
      begin integer X;
            X := READ;
            if ∼(X=0) then REV;
            PRINT(X)
      end;

      begin  {Copy of procedure REV}
            integer X1=5;
            X1 := READ;
→           if ∼(X1=0) then REV;
```

It is clear that we need another copy of REV at the point of call; TPCR and TRR must be applied again, yielding:

```
begin procedure REV;
   begin integer X;
         X := READ;
         if ∼(X=0) then REV;
         PRINT(X)
   end;

   begin     {Copy of procedure REV}
         integer X1=6;
         X1 := READ;
         if ∼(X1=0) then

         begin {Second copy of REV}
               integer X2=6;
               X2 := READ;
→              if ∼(X2=0) then REV;
```

One more copy will do the trick. Note that we have introduced two variables, X1 and X2, and are about to introduce a third variable X3.

```
begin procedure REV;
  begin integer X;
        X := READ;
        if ∼(X=0) then REV;
        PRINT(X)
  end;
```

```
begin  {Copy of procedure REV}
   integer X1=5;
   X1 := READ;
   if ∼(X1=0) then
   begin  {Second copy of REV}
      integer X2=6;
      X2 := READ;
      if ∼(X2=0) then
      begin  {Third copy of REV}
         integer X3=0;
         X3 := READ;
→        if ∼(X3=0) then REV;
```

The zero returned by READ this time indicates an end of tape. The "if" test on ∼(X3=0) fails, so another copy is not needed. We can then unravel the recursive calls, by writing in the remaining statements of the procedure REV:

```
begin procedure REV;
   begin integer X;
      X := READ;
      if ∼(X=0) then REV;
      PRINT(X)
   end;
   begin integer X1 = 5;   {Copy of REV}
      X1 := READ;
      if ∼(X1=0) then
      begin  {Second copy of REV}
            integer X2=6;
         X2 := READ;
         if ∼(X2=0) then
         begin integer X3=0; {Third copy}
            X3 := READ;
            if ∼(X3=0) then REV;
            PRINT(X3)   {"0" is printed}
         end;
         PRINT(X2)   {"6" is printed}
      end;
      PRINT(X1)  {"5" is printed}
   end;
end
```

Notice that REV contains an error; we really don't want to print the end-of-list indicator 0. The procedure should read:

```
procedure REV;
begin integer X;
      X := READ;
      if ∼(X=0) then
      begin REV;
            PRINT(X)
      end
end
```

The point of all this tracing is that a recursive procedure may generate many more distinct variables, or rather, activations of variables, than appear in the source program. Thus X1, X2, X3, ... must be treated as distinct and therefore require separate storage. If the same storage location were used for each activation, the program would simply print a series of N+1 zeroes instead of the intended reversed sequence.

The AOC machine must therefore be prepared to supply an arbitrarily large amount of storage for a running program to accommodate recursive procedures. The data stack in AOC is exactly right for this, and it is easy to see why. The lifetimes of successive activations of any variable will be nested in time as the program runs. For example, we need the storage for X2 after we need it for X1, and we shall cease needing it before we cease needing it for X1. Clearly, a last-in-first-out stack for variables is appropriate.

Our first variable allocation scheme is based upon this observation. We shall arrange that when a block is entered, the previous values of all its local variables will be saved on the top of the current stack. When the procedure exits the old values will be "uncovered" and therefore restored, as the s-pointer is reset to its value before the call.

It is obviously expensive to make a complete copy of a procedure on each run-time call. We shall therefore write the return address on the stack, upon the call. Then we can return at the end to the location indicated by the stack value. If this were not saved on the stack, a recursive call would destroy the return address of the previous call.

We therefore need the following instructions for a procedure call and exit:

```
CP  l:  Call procedure at location l
        s ← s+1
        C(s) ← i {the present location}
        i ← l    {jump to the procedure}

RTN:    Return
        i ← C(s)  {the saved return address}
        s ← s−1
```

The stacking and unstacking of local values can be done by an appropriate sequence of LV's at the block beginning and STD's at its end. We shall develop a more comprehensive plan later.

The above reversal program may then be translated as follows:

```
 0      LC    0   {space for X}
 1      BR    S   {branch around procedure code}
 2  R   LV    X   {save the old value}
 3      READ
 4      STD   X
 5      LV    X
 6      LC    0
 7      EQ
 8      NOT
 9      JIF   L
10      CP    R
11  L   LV    X
12      PRINT
13      STD   X   {restore the old value}
14      RTN
15  S   CP    R
16      HALT
```

Let the input tape contain {5, 6, 0}. Then the stack would appear as in figure 9.2 after the indicated instructions have been executed.

Each successive call of R reserves two locations for the return address (RA) and the previous value of X (shown as X0, X1, X2 in the figure). The return address word is called an *activation record* or *stack marker*. We shall later add more words to the stack marker, to support procedure calls in a more sophisticated manner.

A tentative block translation rule is therefore:

Translate each block independently, allocating space for its variables and its stack marker.

9.6.1. Procedures and the Free Variable Problem

There is an inconsistency between TRR and TPCR as they are presently formulated, and it is called the free variable problem. A *free variable* is a variable referenced in a procedure body but not local to the procedure. That is, it is declared outside the procedure block, but is in a block containing the procedure. For example, variable B in the following program is a free variable relative to procedure P:

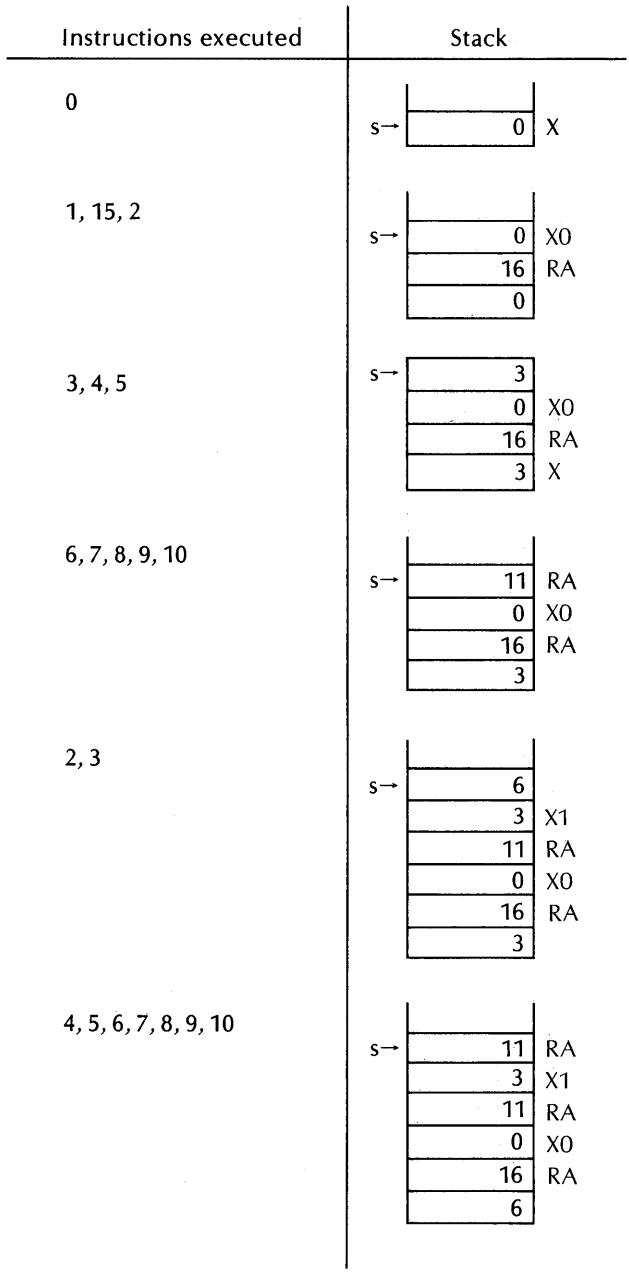


Figure 9.2. Trace of AOC machines through execution of a program with a recursive procedure REV.

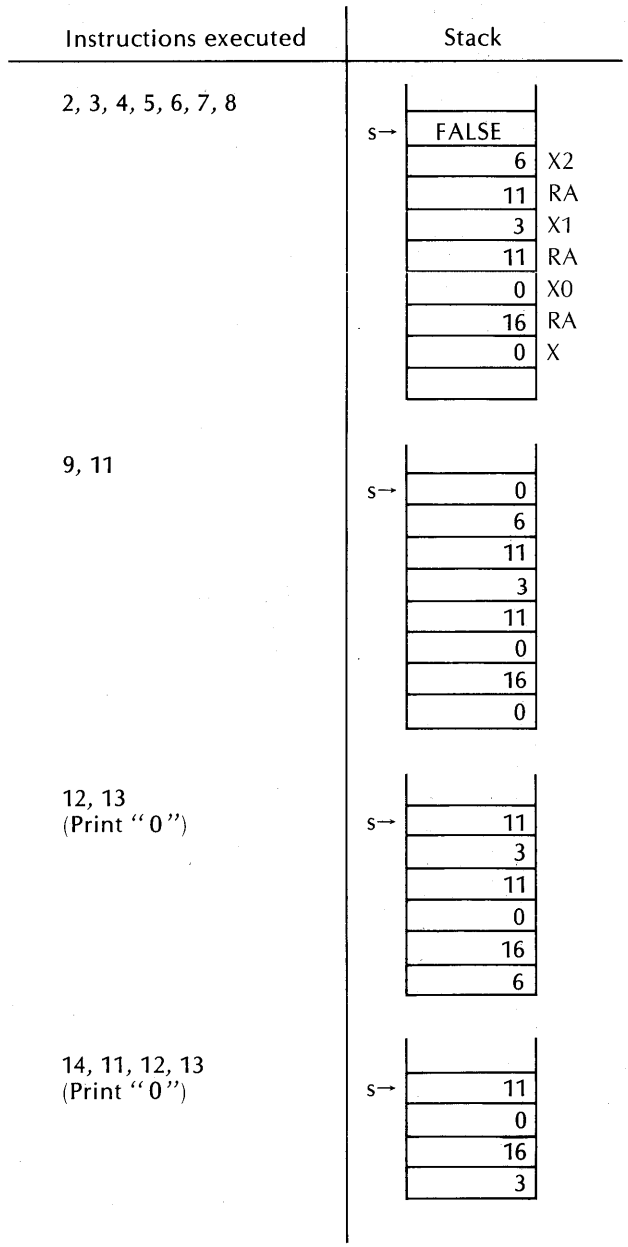


Figure 9.2. (cont'd.)

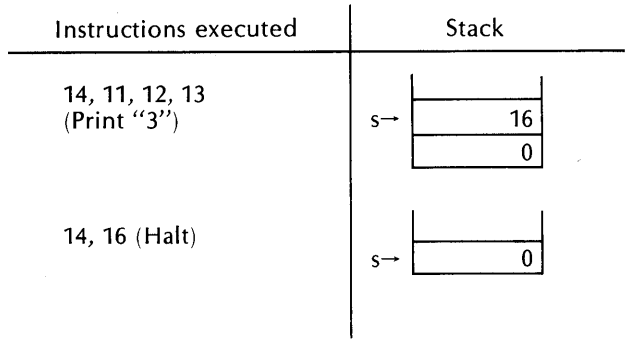


Figure 9.2. (cont'd.)

```
begin integer B;
      procedure P;
      begin B:=5
      end;
      B:=20;
      begin integer B;
            B := 7;
            P;
            print(B)
      end;
      print(B)
end
```

Using TPCR we can arrive at the following program state:

```
begin integer B = 20;
      procedure P;
      begin B := 5
      end;
      B := 20;
      begin integer B=7;
            B := 7;
→           begin B:=5
            end;
            print(B)
```

On the other hand, if we first apply TRR to the second use of B, and proceed as before we arrive at this state:

```
begin integer B = 20;
      procedure P;
      begin B := 5
      end;
      B := 20;
      begin integer B1 = 7;
            B1 := 7;
→           begin B:=5
            end;
            print(B1)
```

The only difference is that the inner block declares B1, not B. It should be clear that the first simulation will print 5, 20, while the second will print 7, 5. TRR has been violated in the first simulation. Our implementation of this procedure as it stands will mimic the first, rather than the second simulation. Why this happens is not hard to see. When procedure P is called, its variable environment consists of the second B (B1), and this environment is not changed by the call mechanism. However, as the procedure appears in the source, its variable environment should consist of the first B, not the second, in order to preserve TRR. We see that a procedure call requires adjustments to the variable environment of the stack.

We clearly must also resolve the inconsistency between TPCR and TRR. We know that TRR is an important rule to have. It means that the writer of a procedure need only pay attention to the variables within his procedure and those in blocks that contain his procedure. He should not have to be concerned with the many and varied variable environments of the procedure calls. We therefore amend TPCR to require that it may not be applied until all independently declared variables have been made distinct through TRR.

Even this change is not good enough; it seems to break down for recursive procedures which seem to require application of TPCR first (to copy the procedure), then TRR. The situation gets quite complicated (see Kanner, in Rosen [1967], pages 228-252). We choose not to further develop the copying rules, but rather embark upon a revision of the procedure calls, blocks, and variable accessing instructions in order to preserve TRR.

9.6.2. Textual Addresses

We need a special naming system for variables in Algol programs. The *block level* (BL) of a block is the number of blocks outside it. The outer block therefore has BL = 0, the one immediately inside it has BL = 1, etc. The *textual address* (TA) of a variable is a pair of integers [B,j], where B is the BL of the block in which it was declared and j is an offset (measured in stack words) from a stack marker corresponding to the block of level B. For

programs with only simple variables and fixed-size arrays, j can be determined at compile time. The offset j starts with 0 for the first variable. Block level B is known by the compiler, so the TA of every variable can be fixed at compile time.

For example, consider the program

```
begin integer A;
      procedure P;
      begin integer X,Y;
            procedure Q;
            begin integer Z;
                  Z := X;
                  X := X + A;
                  Y := X + Z
            end;
            X := A;
            A := A + 1;
            Q
      end;
      A := 2;
      P;
      begin integer C;
            C := A;
            P;
            print(C)
      end
end
```

Then the TA's of the variables are as follows:

```
A  [0,0]    e.g., block 0, word 0
X  [1,0]    e.g., block 1, word 0
Y  [1,1]    e.g., block 1, word 1
Z  [2,0]
C  [1,0]    Note:  Same as X
```

TA's will be used, in conjunction with the display memory, to find the locations of variables at run-time. The rule is relatively simple:

The address of the proper activation of a variable X with TA = [B,j] is D(B)+j.

Recall that D(B) is the contents of the B'th display register. The display register will be used to keep track of the locations of the various variable

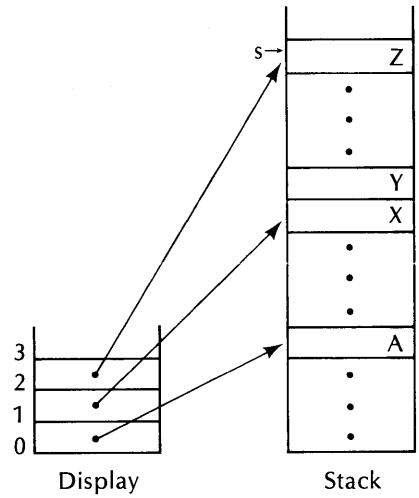


Figure 9.3. Stack containing (among other information) four local variables (A, X, Y, and Z), and the display pointers. On this run-time configuration, a procedure at block level 2 is being executed.

domains in the stack. Those variables in block B will be in the stack relative to D(B).

For example, while the statement Z:=X in the above program is being executed, the display and stack will appear as in figure 9.3. This scheme clearly permits other variables to be stored in the spaces between D(0), D(1), and D(2).

We now redefine the addressing instructions as follows:

LV B,j: Load value
$s \leftarrow s+1$
$C(s) \leftarrow C(D(B)+j)$

LA B,j: Load address
$s \leftarrow s+1$
$C(s) \leftarrow D(B)+j$

STD B,j: Store direct
$C(D(B)+j) \leftarrow C(s)$
$s \leftarrow s-1$

9.6.3. Block Entry and Exit

We need another change in our implementation. Rather than stacking the old values of variables when a block is entered (as before), we reserve space in

the stack for the new activations of variables whenever a block is entered. We need two new operations for this, BE and EB. These operations will have to be expanded later, but for now:

```
BE  B,n:  Block entry at level B, with n variables
          D(B) ← s+1      set up display element
          s ← s+n         reserve n words

EB  n:    Exit block with n variables
          s ← s−n         release n words
```

The compiler will always know the appropriate B and n values to use for these instructions. Also note that the display-top variable b is not involved.

BE can be used instead of INCS n for most purposes. In Algol, arrays can only be declared upon entering a block. If they have fixed dimensions, then the total space needed for them and the simple variables can be worked out by the compiler. All the space may then be allocated by a single BE instruction.

Arrays with variable dimensions (dimension determined at run-time) must be allocated dynamically on the stack or in the heap through special AOC instructions. However, a dope vector of fixed length can be allocated by the compiler. A complete discussion of dynamic allocation is given later.

Example.

The translation of the above program is then:

```
        BE   0,1   begin integer A {block 0, 1 word}
        BR   STRT  {branch around procedure code}
                   procedure P
P       BE   1,2   begin integer X,Y {block 1, 2 words}
        BR   PSTR  {branch around procedure Q code}
Q       BE   2,1   procedure Q begin integer z
        LV   1,0   Z := X
        STD  2,0
        LV   1,0   X := X+A
        LV   0,0
        ADD
        STD  1,0
        LV   1,0   Y:=X+Z
        LV   2,0
        ADD
        STD  1,1
        EB   1     end
        RTN
```

```
PSTR    LV   0,0   X := A {start of procedure P code}
        STD  1,0
        LV   0,0   A := A+1
        LC   1
        ADD
        STD  0,0
        CP   Q     Q
L3      EB   2     end
        RTN
STRT    LC   2     A := 2 {start of outer block program}
        STD  0,0
        CP   P     P
L1      BE   1,1   begin integer C
        LV   0,0   C := A
        STD  1,0
        CP   P     P
L2      LV   1,0   print(C)
        PRINT
        EB   1     end
        EB   1     end
        HALT
```

This program will show a flaw in our design. Let us simulate it. After the first call of P and execution of its BE, the display and stack appear as in figure 9.4.

After the call of Q and its BE operation, the display and stack appear as in figure 9.5. Immediately after the Q return, the display and stack appear as in figure 9.6.

The return from P just moves the s-pointer to A's location again. After the block for C, P, and Q have been entered again, the display and stack appear as

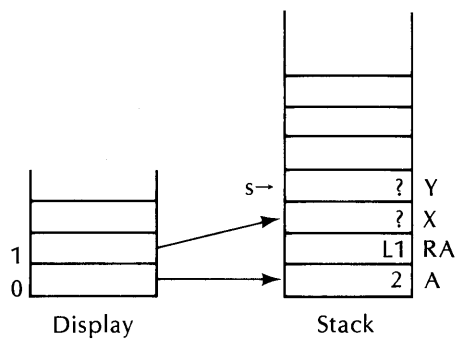


Figure 9.4. A configuration during procedure execution (see text).

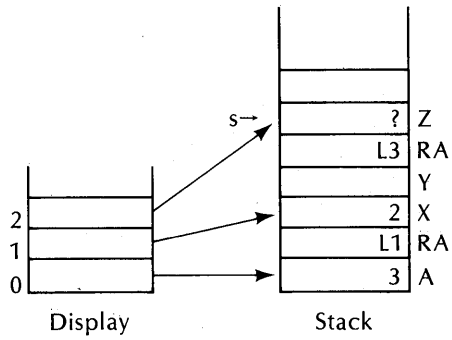


Figure 9.5. A configuration during procedure execution (see text).

in figure 9.7. Notice that there is no display entry pointing to C's location, as is required by TRR from the environment of Q. Instead, the environment seems to be that of the block containing Q's call, namely variables A, X, and Y. Of course, C cannot be accessed from within P, however, it should be accessible from within Q, and it isn't.

Now Q stores A into what it thinks is C. However, the display is such that A is actually stored into X. After procedures P and Q have run to completion and returned, the instructions

```
LV    1,0
PRINT
```

will print the value of X, not C, as they were supposed to. The stack configuration after exiting from Q and P is shown in figure 9.8. Register b is 0 and most of the stack has been dropped.

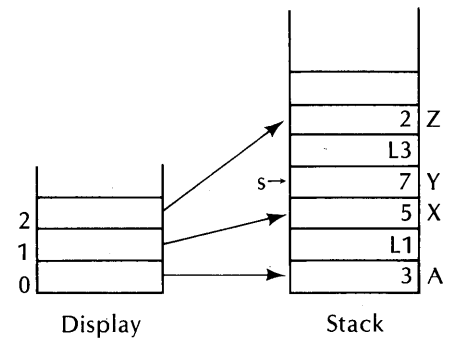


Figure 9.6. A configuration during procedure execution (see text).

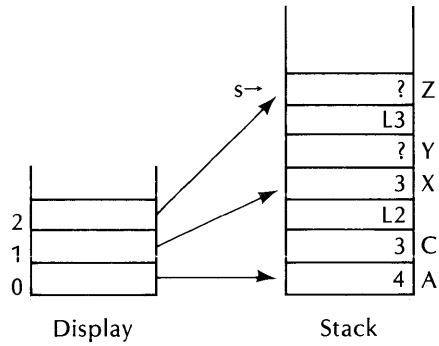


Figure 9.7. A configuration during procedure execution (see text).

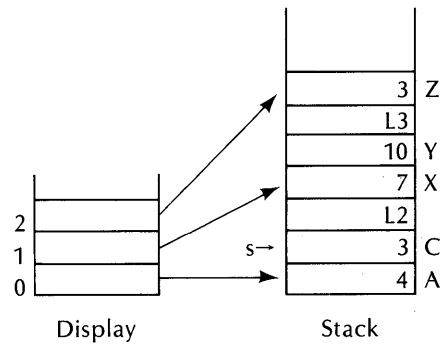


Figure 9.8. A configuration during procedure execution (see text).

The essential problem is that the display entries of a block are lost if a call at block level m is to a procedure at block level n, where n < m. Such a call is called an *up-level call*. We clearly must set the display correctly on an up-level call to correspond to the environment of the called procedure (BL = n), then reset it (to BL = m) upon exit.

9.6.4. The Static Display Chain

We shall keep a complete copy of the current display in the stack, as well as previous ones which will be needed later, tucked away among the variables. These copies are called *static chains* because they mirror the display, but are not actively involved in addressing variables.

We shall also need register b, which always contains the BL of the block whose code is executing; b also serves as a stack-top index for the display.

We shall arrange that the following is always true, for i = 1,2, ..., b:

$$C(D(i)-1) = D(i-1)$$

This relation is shown in figure 9.9 for b=2. Note that the top display element D(b) is not in the stack and that the element i is just below the location pointed to by D(i−1). Also, C(D(0)−1) can contain anything–this word is never accessed by the display manipulation system.

Since BE is the only operation that changes the display, we assign it (and EB) the task of building and maintaining the static chain.

```
BE  n:  Block Entry with n variables
        C(s+1) ← D(b)      save static pointer
        b ← b+1             increment block level
        D(b) ← s+2         set display element
        s ← s+n+1           reserve storage
EB:     Exit block
        s ← D(b)−2         reset s
        b ← b−1             decrement block level
```

Notice that since b is used, BE no longer needs its B parameter, and EB can reset s without knowing how many variables were introduced.

In the absence of procedure calls, the block index b moves up and down in an orderly fashion as blocks are entered and exited. An up-level procedure call introduces a discontinuity: the block index b jumps downward by 1 or

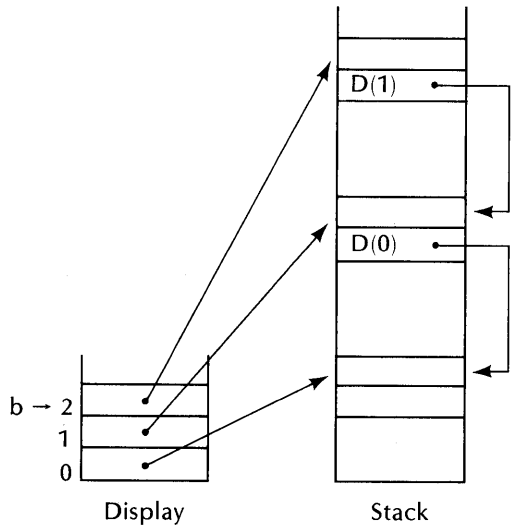


Figure 9.9. The static chain. This contains all the display information in the stack, except the topmost cell D(b).

more. To allow RTN to reverse this jump, we must save additional information on procedure entry, namely the values of b and D(b) at the time of the call.

We therefore stack i (the return address), b, and D(b) on a call. This triple of quantities is called a *transfer point* (TP). In this context, we may think of it as a sort of generalized return address. Now we define PE and redefine CP and RTN.

```
CP  l:  Call procedure at location l
        s ← s+1
        C(s) ← D(b),b,i      save TP
        i ← l                  transfer

PE  B:  Procedure entry, block level B
        b ← B            adjust b
```

The block level B is the level of the block in which the procedure is declared. Clearly, the compiler must save this in its symbol table associated with the procedure.

```
RTN:    Return
        Go(s)       adjust display and transfer (sets i)
        s ← s−1          reset s
```

Go is a procedure that accepts a transfer point location and carries out the transfer by resetting i, b, and the display. Go is given below:

```
procedure Go(t: integer);
begin  {resets display}
  var j: integer;

  D(b),b,i := C(t);  {unstack;
     note that b is filled before D(b)}
  j := b;
  while j>0 do
  begin
    D(j−1):=C(D(j)−1);
    j:=j−1
  end
end
```

Usually, only some of the display elements need to be reset. If b′ is the number of the smallest block which textually encloses the two points being transferred between, then D(0), ..., D(b′) are already correct and the b′

assignments in the loop of Go are redundant. One could therefore reduce the number of repetitions of the loop.

The order of the instructions emitted is essential. On a procedure entry, the code should be:

```
PE  B
BE  n
```

On a procedure block end, the code should be:

```
EB
RTN
```

Let us now simulate the previous program in the light of our new operations. The only modifications to the program are that the instructions PE 0 and PE 1 should be inserted at the beginning of P and Q respectively, since these are the levels at which P and Q were declared. Further, the block levels can be dropped from the BE's and variable counts from the EB's.

After entering the block in which C is declared, the display/stack configuration is as shown in figure 9.10. Just before the EB in Q is executed, the configuration is as shown in figure 9.11. After execution of the EB and RTN of Q and EB of P, the configuration is as shown in figure 9.12. After executing the RTN of P, which restores the display, the configuration is as shown in figure 9.13. We see that the 7 was properly stored in X, not C, and that C's value of 3 is preserved through the calls.

Thus D(1) has been restored to its value from before the call of P.

We have now completed our version of the procedure and block operations. Although the example presented did not involve them, the following features of Algol require this complicated implementation:

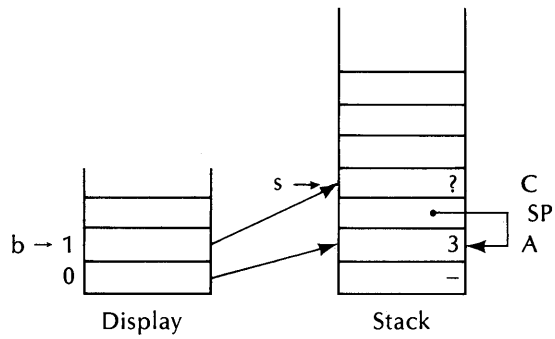


Figure 9.10. Use of the static display at run-time (see text).

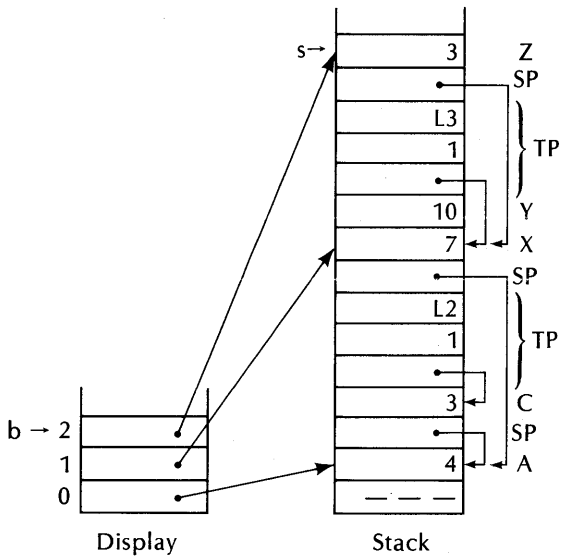


Figure 9.11. Use of the static display at run-time (see text).

1. The renaming rule, which requires that the values of free variables of a procedure are determined by the textual position of the procedure declaration
2. Recursive procedures, which require more than one activation of the same variable to coexist in general.

9.6.5. The Display Revisited

The display is actually redundant now, since the same information is contained in the static chain. One may therefore consider omitting the display in an implementation.

Wirth [1971a] found that three display registers were sufficient for his Pascal compiler source, and their access frequencies are as follows:

D(0): 84.2%
D(1): 14.7%
D(2): 1.1%
D(3): Never used

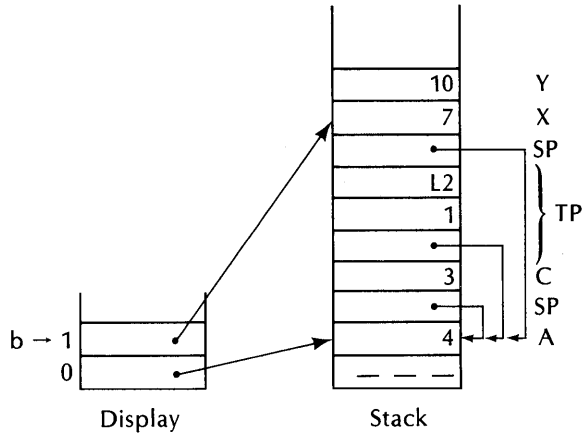


Figure 9.12. Use of the static display at run-time (see text).

This means that the Pascal compiler never exceeded a nesting level of 2 in its procedures. However, Pascal is not fully block-structured; that is, a BEGIN END pair cannot be used by itself to declare new variables, as in Algol 60, PL/I, and Algol 68. In these languages, one might expect more nesting levels to be used.

Given more nesting levels, it appears from Wirth's study and from other observations that the most frequently accessed levels are the "b" level (most

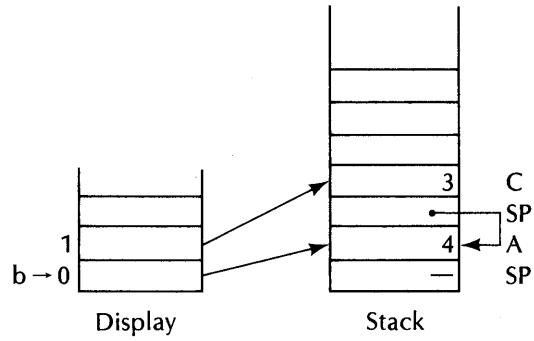


Figure 9.13. Use of the static display at run-time (see text).

local) and the "0" level (global). These levels account for most of the variable references. One can then justify a display consisting of just two registers, one for global references and another for local references. The few intermediate level references can be handled in either of two ways:

1. For every intermediate level reference, search down the static chain to the appropriate level and access the data through the indicated stack location.
2. Locate all intermediate level references in a first pass through a procedure. Then generate local addresses to each variable referenced at an intermediate level in the procedure (by following the static chain). Every data reference then is either local direct, local indirect, or global. Note that two passes are needed in this approach, since the stack allocation of local indirect labels cannot be made in the middle of expression evaluation, and a free variable can show up anywhere.

Of course, if the display is implemented in hardware, and the necessary display manipulation in firmware, as in the Burroughs 5500, its overhead cost in execution time and code is negligible.

9.6.6. Labels and GOTO's

Any statement in an Algol program may be labeled. The scope of the label is the smallest textually enclosing block. A label is similar to a procedure in branching, except that a branch to a label carries no parameters and no return is expected. A branch B to some label L is legal only if the level of B is greater than or equal to the level of L, just as for a procedure.

A GOTO must reset the i, b, and s pointers to the context of the label. We must modify BE and EB once more to save the s pointer for this purpose. The saved quantity is called the *working pointer* (WP).

```
BE  n:  Block Entry with n variables
        C(s+1) ← s+n+2     save WP
        C(s+2) ← D(b)      save the static pointer
        b ← b+1
        D(b) ← s+3
        s ← s+n+2

EB:     Exit block
        s ← D(b)−3
        b ← b−1
```

Now we define an operation analogous to PE, intended to precede any branch to a label in a different block level:

```
LE  B:  Label entry in block at level B
        b ← B
        s ← C(D(b)−2)
```

Notice that LE is a "no operation" effectively if the branch and the label are in the same block, hence could be omitted as an optimization in this case. Also, when a branch is executed, the stack configuration is expected to be exactly as it was upon entering the block containing the label, with no additional words. This means that a branch within an expression evaluation (Algol 68 permits such a thing) will cause any partial expression results on the stack to be lost. Even this loss is OK, inasmuch as branches are to complete statements, not to points within expressions.

An Algol 60 FOR loop should be implemented by stacking the step size and limit. A branch from within a FOR loop will lose this information, even if the target of the branch is within the loop. For this reason, a BE-EB pair should be executed just covering the loop code. Then a branch within the FOR loop, or even out of the loop into some covering loop, will adjust the stack properly.

The JP and JIF instructions from section 9.2.3 may still be used for the GOTO statement.

Consider the following program:

```
begin integer A;
      A:=0;
      begin integer B;
            procedure G;
            begin B:=READ;
                  if B<0 then goto ENDF;
                  A := A+B
            end;
            LOOP:  G;
            goto LOOP
      end;
      ENDF: PRINT(A)
end
```

Its translation is:

```
BE      1     begin integer A
LC      0     A:=0
STD     0,0
BE      1     begin integer B
JP      L2
```

```
G   PE      1     procedure G
    BE      0     begin
    READ          B:=READ
    STD     1,0
    LV      1,0   if B<0 then
    LC      0
    LS
    JIF     L1
    LE      0
    JP      E     goto ENDF
L1  LV      0,0   A:=A+B
    LV      1,0
    ADD
    STD     0,0
    EB            end
    RTN
L2  CP      G     LOOP: G
    LE      1
    JP      L2    go to LOOP
    EB            end
E   LV      0,0   ENDF: PRINT(A)
    PRINT
    EB
    HALT
```

Notes:

1. The first JP in the program allows us to write the translation of G in the same relative position as its procedure declaration.
2. The BE-EB pair around G's translation could be omitted since no new storage is needed.
3. The EB immediately before statement E is never executed.

Exercises

1. The instructions CP 1 and RTN cause a branch to some other program memory location. Exactly how is this achieved?

2. Assign textual addresses to each of the variables in the following program, then translate it to AOC code:

```
begin integer A;
   integer array B[5:9],
                 C[-3:2];
   integer D,E;

   procedure SAM;
   begin integer B,C;
      integer array X[0:15,3:7];

      procedure ED;
      begin integer array Q[3:9];
         integer Z;
         Z:=X[3,6];
         B:=Z+1
      end;

      procedure MAX;
      begin integer array B[0:5];
         integer Y,Z;
         Z:=B+1;
         ED;
      end;
      B:=C+3;
      if B<0 then go to ENDP;
      MAX;
   end;
   A:=B+C[B]+D-E;
   SAM;
ENDP: end;
```

3. Assume that the display D is replaced by a single register Q that effectively contains D(b) at all times. (Register b still exists.)

   (a) Redefine each of the instructions that involve D and procedure Go, to reflect this plan.

   (b) Give an algorithm for up-level address references, assuming that each reference requires a static chain sequential search.

4. One might suppose that procedure Go could be translated into AOC instructions, and that it could operate using the display, stack, etc. Is this so? If not, why not? Could it be written in a mixture of AOC instructions and "contents" notation, to operate on the current stack?

5. Show that LE is a "no-operation" if the branch and the label are in the same block.

6. Given that some branch does not require an LE, can this fact always be detected in a one-pass compiler? In a two-pass compiler?

7. Design instructions to implement an Algol FOR loop. The loop rules form is

   for <var> := <expr1> [step <expr2>] until <expr3>
   do <statement>

   The loop rules are

   (a) The step size <expr2> may be positive or negative, but not zero. If the step is omitted, a step size of +1 is assumed.

   (b) The <statement> cannot affect the step size or termination condition <expr3>, but can affect the <var> value.

   (c) The test of <var> value against the termination value <expr3> is made before the <statement> is executed; thus the <statement> may not be executed.

   (d) The <statement> is executed on any iteration only if the current values satisfy

   $$\text{SIGN}(\langle\text{expr3}\rangle) \times | \langle\text{expr3}\rangle - \langle\text{var}\rangle | \geq 0$$

   where SIGN(X) = if X > 0 then +1 else −1, and where |X| = (if X>0 then X else −X).

   (e) The <statement> may contain any Algol statement or block, including other FOR loops.

8. Give a reasonable set of restrictions for branches into and out of FOR loops (cf. exercise 8 above), and sketch a compiler algorithm to detect such restrictions. What stack adjustments, if any, are needed on such branches?

9. When are both a PE and a BE required on a procedure call? Give examples of programs in which (a) only a PE is needed, and (b) only a BE is needed upon entering some procedure or block.

10. Pascal permits the declaration of variables only upon a procedure entry, not upon an arbitrary block entry (BEGIN END). Are separate PE and BE instructions needed? If not, design a combined instruction.

9.7. Arrays

Arrays can be implemented in at least two different ways. The first of these, *packed* arrays, conserves memory space at the expense of element access time. The second, *matrix pointer* arrays, requires more memory space, but provides a rapid access means.

Both systems require a *dope vector* that can be allocated by the compiler, and assigned to a transfer point [B,j]. The dope vector structure is the same for both array access means and is shown in figure 9.14. It consists of $2n+1$ words, where n is the number of array dimensions (the *rank*.) Although the upper and lower limits of an array may be altered dynamically, its rank is fixed at compile time.

9.7.1. Packed Arrays

Packed arrays are implemented through two new instructions, MAS and AVA, as follows:

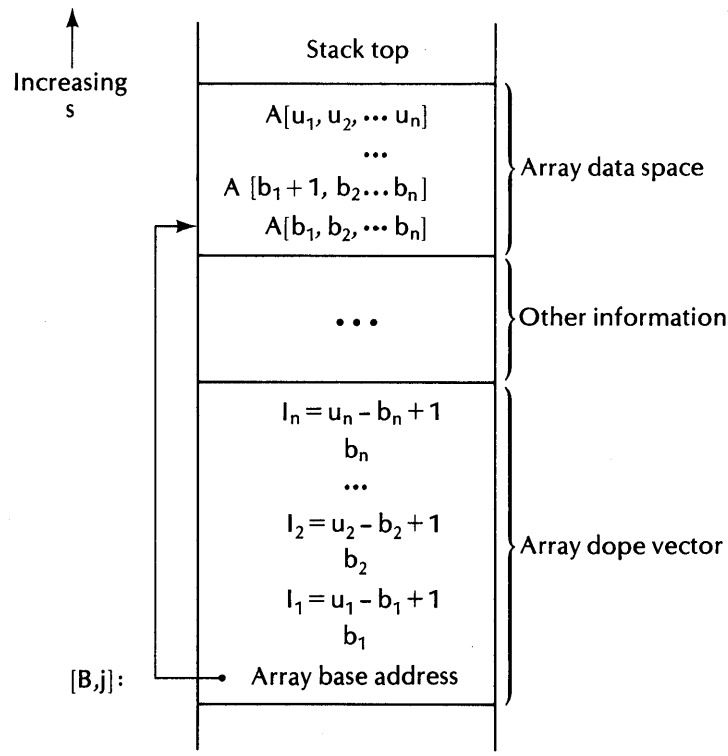


Figure 9.14. Stack configuration upon completing allocation of a packed array through procedure MAS. The array dope vector is given, then MAS allocates the necessary space and sets the array base address in the dope vector.

MAS B,j,n: Make array space.
[B,j] is the array base address,
n is the array rank.

MAS allocates space for the array from the stack, based on the size and lower bound information given in the dope vector. The array base word address is also filled in by MAS.

AVA n: Array Variable Address.

AVA will calculate the address of an array element, given its subscripts and the array base word address arranged on the stack. AVA may also be used to check that the given subscripts lie within the dimensioned range of the array.

Array allocation occurs in two steps as follows:

1. While scanning the declarations in the head of a block, the compiler allocates space for the dope vector of each array. Code may be emitted to generate the lower bound (b_i) and size (l_i) information in the vector (see figure 9.14). The vector is arranged such that a pair (b_i, l_i) may be computed and left behind on the stack, while parsing a declaration, from left to right, to form a dope vector. The textual address of the dope vector [B,j] and the rank n are noted and kept as array attributes in the compiler's symbol table. No space is allocated yet.
2. After all the declarations are scanned, MAS is called for each of the arrays. The allocated space will depend on how the dope vector is filled in during execution of the block-head code, hence the locations and size of the array spaces (except the location of the first) will be unknown to the compiler.

In general, the dope vector and the allocated array space will be separated by an unknown number of words. Only the dope vector transfer point is known to the compiler.

Array space and the dope vector created for the array declaration

$$A[b_1{:}u_1, b_2{:}u_2, \ldots b_n{:}u_n]$$

by MAS is shown in figure 9.14. For dimension i, b_i is the least value that a subscript may take and u_i is the greatest value that subscript i may take. These may be expressions in general, containing variables previously declared and defined.

An array's data space is allocated somewhere higher in the stack than its dope vector. The array elements start with

$$A[b_1, b_2, \ldots b_n], A[b_1 + 1, b_2, \ldots b_n] \ldots$$

and end with

$$A[u_1, u_2, \ldots, u_n]$$

Note that the left-most index is the most rapidly varying one. We shall see that this arrangement leads to an efficient implementation.

The array base word is needed to give the array a constant transfer point despite the varying locations and sizes of the array elements. The base word is filled in during execution of its MAS instruction.

For example, consider the program

```
begin integer N;
          N:=READ;  {dimension is variable}
      begin integer I;
          integer array A[1:N],
                        B[3:5,N + 1: N + 15];
          .
          .
          .
```

This program translates to:

```
BE   1    begin integer N;
READ      N:=READ
STD  0,0
BE   1    begin integer I;
            {"1" is the words needed for
            variable I.
             The dope vectors for A and B will be
             constructed on the fly.
             The textual addresses are:
              I: [1,0],
              dope vector A: [1,1],
              dope vector B: [1,4] }
LC   0    {dope vector A base address,
            not known yet}
LC   1    {A lower limit}
LV   0,0  {size, N}
LC   0    {B base address, fixed up later}
LC   3    {B lower limit, b1}
LC   3    {first dimension size, l1}
LV   0,0  {N}
LC   1    1
ADD       {lower limit of B, b2}
LV   0,0  {N}
LC   15
ADD       {upper limit, u2}
```

```
LV   1,7  {lower limit}
SUB
LC   1
ADD       {size, l2}
MAS 1,1,1 {allocate space for A}
MAS 1,4,3 {allocate space for B}
.
.
.
```

The MAS procedure is defined as follows. It requires the transfer point [B, j] of the dope vector and the number of dimensions N. It then allocates space for the array and sets the array base address in its dope vector.

```
procedure MAS(B, j, N: integer);
begin  {make array space}
   {[B, j] = TA of dope vector,
    N = array rank}
   var DV,T,I,X: integer;
       {temporary variables}
   DV:=D(B)+j;  {absolute dope
        vector address}
   C(DV):=s+1;  {base address}
   X:=1;
   DV:=DV+2;

   for I:=0 until N-1 do
   begin
      X:=X*C(DV); {l1 × l2 × ... × ln}
      DV:=DV+2
   end;
   s:=s+X    {the space allocation}
end
```

MAS is designed to execute at run-time. However, it cannot use the data stack of the user program without some changes. If MAS is called as a procedure, a stack marker will be established, yielding a new stack top address s. The array will be allocated using this new s, after the temporary variables of MAS have been allocated. Unfortunately, upon returning, all the allocated space will disappear, through the way the procedure return works.

If it is necessary that MAS operate on the user stack, then (1) the base address should be adjusted to reflect the known number of words added to the stack by the MAS call, and (2) just after the stack space is allocated, the MAS stack marker and its temporary variables should be moved to the top of the

allocated space, and D(B) adjusted. Then upon exit, the allocated space will in fact be left behind in the stack as wanted, and all the traces of the MAS call will have gone away.

Now consider the location of an array element:

$$A[i_1, i_2, i_3, ..., i_n)$$

This array element is stored in the stack location

$$\text{(base location)} + i_1 - b_1 + l_1*(i_2 - b_2 + l_2*(i_3 - b_3 + \ldots + l_{n-1}*(i_n - b_n) \ldots))$$

The AVA instruction expects the following list on the stack: $i_1, i_2, \ldots, i_n$, DVA, where DVA is on the stack top, and is the dope vector address. The necessary code sequence to access the array elements is therefore:

```
{evaluate i1}
{evaluate i2}
    .
    .
    .
{evaluate in}
 LA  B,j  {dope vector transfer address}
 AVA n       {n=array rank}
{array element address is
   left on the stack}
```

The following procedure defines AVA. This, like MAS, needs some modifications if it is to operate on the user stack. We leave the necessary modifications to an exercise.

```
procedure AVA(N: integer);
begin   {compute array element address
         from stack information}
   var T,U,I: integer;

   T:=C(s);  {dope vector address}
   T:=T +2*N -1;  {address of bn}
   s:=s-1; {drop transfer address
            from stack}
   U:=C(s)-C(T);  {element offset for
                   one-dimensional array}
   I:=N-1;

   while I>0 do
   begin
      s:=s-1;
```

```
        T:=T-1;
        U:=U*C(T) + C(s) - C(T-1);
        T:=T-1;
        I:=I-1
    end;
    C(s):=U+C(T-1) {final element address}
end
```

In AVA, I is used as a counter to step through the dope vector, T is an address into the dope vector, and U is an element offset variable that becomes the offset from the base address of the array at the end of the while-do. Some optimizations of the while loop are suggested, since the dope vector elements are needed in just the order that they appear, moving downward in the stack.

We leave as an exercise a proof that AVA correctly computes the array element address.

Example

Here is a program and its translation that illustrates procedure AVA.

```
begin integer N,M;
    N:=READ;
    M:=READ;
    begin integer array A[1:N],
            B[0:M, N:20, N:M+N];
        integer I,J;
        I:=READ;
        J:=READ;
        A[I] := I+J;
        B[J,19,I+3] := A[J-1]
    end
end
```

```
    BE   2    begin integer N,M
    READ      N:=READ;
    STD 0,0
    READ      M:=READ;
    STD 0,1
    BE   0    begin ... A[], B[], I, J;
                {space will be allocated as we move along.
                 The final transfer points will be:
                  A d.v. 1,0
                  B d.v. 1,3
                  I      1,10
                  J      1,11}
```

```
LC   0     base address for A
LC   1     store base, b1=1
LV   0,0   fetches N, is in fact the wanted size
LC   0     bottom of descriptor words for B
LC   0     b1 = 0
LV   0,1   fetch M
LC   0
SUB
LC   1
ADD        u1 - b1 + 1 = M+1
LV   0,0   b2 = N
LC   20    u2
LV   0,0
SUB
LC   1
ADD        u2 - b2 + 1 = 21-N
LV   0,0   b3=N
LV   0,1
LV   0,0
ADD
LV   0,0
SUB
LC   1
ADD        u3 - b3 + 1 = M+1
INCS2      space for I, J
MAS 1,0,1  create space for A
MAS 1,3,3  create space for B
READ       I:=READ;
STD 1,10
READ
STD 1,11   J:=READ
LV   1,10  I
LV   1,0   array base word for A
AVA        get address of A[I]
LV   1,10  I
LV   1,11  J
ADD
ST         A[I] := I+J
LV   1,11  J
LC   19
LV   1,10  I
LC   3
ADD
LV   1,3   array base word for B
```

```
AVA       get address of B[...]
LV  1,11
LC  1
SUB
LV  1,0   array base word for A
AVA       get address of A[J-1]
CONT      get contents of A[J-1]
ST        B[J,19,I+3] := A[J-1]
EB
EB
HALT
```

9.7.2. Array Access Through Matrix Pointers

Given an array of rank n, we may organize the array access by constructing *pointer vectors* in the stack, as well as allocating space for the array elements. The determination of an array element address is much faster with pointer vectors than through the index calculations of AVA above, if the rank is two or more. If the rank is one, the two methods are essentially the same.

Let us begin with access. For rank n, and an array element of the form

$$A[i_1, i_2, i_3, \ldots, i_n]$$

we construct a dope vector exactly as before at the transfer point [B,j]. (However, the base address and allocation is handled by procedure MMAS, given below.) Then the following code sequence is emitted:

```
LV  B,j   {this will be the
            base address, less b1}
{evaluate i1}
ADD       {end of one index}

CONT      {for a second index}
{evaluate i2}
ADD       {end of second index}

  .
  .
  .
CONT      {for the nth index}
{evaluate in}
ADD       {element address will be
            on the stack}
```

The simplicity of this scheme speaks for itself. Instead of building a list of

indices, then calling AVA, the simple instructions CONT (indirection) and ADD are used to determine the address.

A conceptual diagram of this access scheme is given in figure 9.15. Only one dope vector element, the base address, is needed. The base address of the dope vector points to a fictitious memory location, b_1 words below a vector of l_1 pointers. Note that the first index i_1, must be at least b_1, and is added to the base address during access. After the first ADD, we therefore have the address of one of the l_1 pointers. CONT fetches its contents, and this is the address of another fictitious memory location, b_2 words below a vector of l_2 words. At the end of the access sequence, we have the address of an element.

A specific stack configuration for the declaration

var X: array [2:3, 3:2] of integer

with the dope vector X address 11 and the top of stack address 20 upon calling procedure MMAS, is shown in figure 9.16. The dope vector is in locations 11/15, the first (and only) matrix pointer vector is in locations 21/23, and the

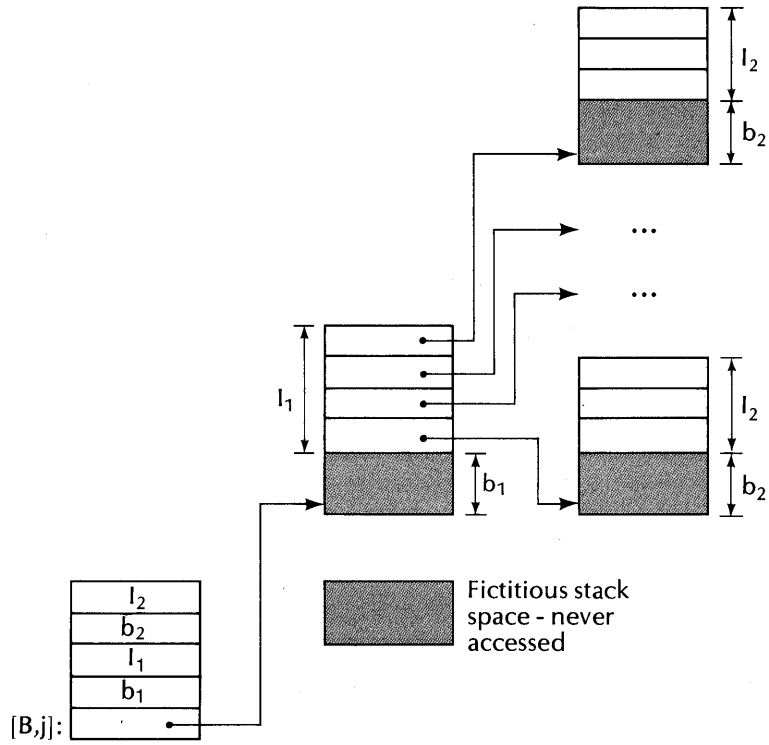


Figure 9.15. Conceptual stack configuration for matrix-pointer array access.

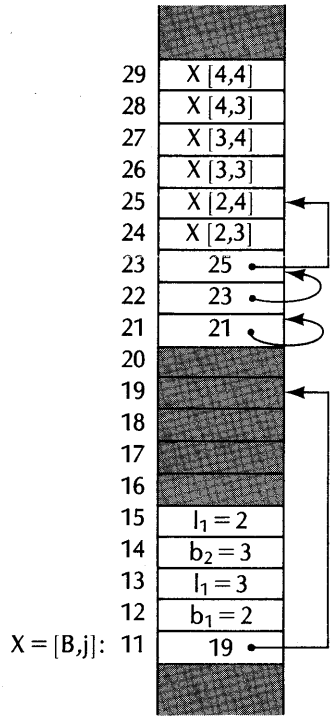


Figure 9.16. Actual stack configuration for matrix-pointer array access, just after MMAS has allocated the necessary space and set up the pointers.

data elements are in locations 26/29. The configuration is exactly as MMAS, given next, would generate.

Now let us define procedure MMAS. This sets the base address for an array, allocates space for it, and sets the pointer vectors appropriately.

```
procedure MMAS(B, j, N: integer);
begin  {make matrix array space.
        [B, j]: dope vector
                transfer address,
        N: array rank}
   var DV: integer;

   procedure SETP(N,A,D: integer);
   begin {set pointers}
      var I: integer;

      if N>0 then
      begin  {return if N is zero}
         for I:=0 until C(D-1)-1 do
```

```
        begin
            C(A+I):=s+1-C(D);
                {set one pointer}
            s:=s+C(D+1);  {allocate a
                vector}
            SETP(N-1, s-C(D+1)+1, D+2)
                {then fill the vector}
        end
    end
end;  {of procedure SETP}

DV:=D(B)+j;  {absolute dope
    vector address}
C(DV):=s+1-C(DV+1);
    {base address, less offset b1}
DV:=DV+1;
s:=s+C(DV+1);    {space for a vector}
SETP(N-1,s-C(DV+1)+1, DV+2)
    {fill the space}
end  {of procedure MMAS}
```

Procedure SETP is the key to understanding MMAS. When SETP is called, it expects that a contiguous array of words, starting at address A, has been allocated on the stack, but not filled. It also expects D to be the address of the next dimension pair in the dope vector. The purpose of the FOR loop is to fill this vector with the addresses of spaces that SETP is allocating. Note that when N reaches 0, the space at address A is simply allocated and not filled; this becomes data space.

If the FOR loop instructions are executed once or more times, each iteration places one address in the vector. The address is the next available stack location s+1. After the address is placed, space is allocated for the vector (or data) and SETP is called to fill it, based on the next dimension (at location D+2). The recursive calls of SETP effectively set up an entire matrix of pointers and array spaces before returning to set up the next pointer in its list. At the end of a set of recursive calls is an allocation of uninitialized array space. The maximum depth of recursive calls is r, the number of dimensions of the array. However, the total number of calls is $n_1 \times n_2 \times \ldots \times n_{r-1}$, or 1 if r=1. This of course can be a large number of calls. However, MMAS is only called upon a declaration. Access of an array element is fast inasmuch as no multiplications are needed.

MMAS will result in sets of data space and pointer vectors intermixed in the stack in some fashion. It is obviously essential that array indices lies within their declared bounds, otherwise data values will be treated as pointers, or pointers will be overwritten with data. Clearly, array access can

be embellished with array bounds tests, although at some increase in access code and execution time.

Procedures MMAS and SETP expect their parameters to be passed by value. This means that a local copy of the parameter value is built on a stack. The copy may therefore be used and modified without affecting the original value. The FOR loop in procedure SETP must carry its limit value on a stack, as well. Of course, the local variable I in SETP is stacked.

MMAS cannot be executed on the user stack without some modifications to prevent loss of the allocated space and pointers upon return. We leave details to an exercise.

9.7.3. Dynamic Arrays and Redimensioning

An Algol 60 variable may be designated OWN or be considered local by default. A local array is allocated by one of the previous schemes upon entering a new block and is deallocated upon exiting from the block. Deallocation is automatic—as the stack top pointer s is simply reset to its value upon entering the block. There are also no language features that permit the dimensions of a local array to be changed within the block (at least not in Algol or Pascal. However, PL/I permits arbitrary redimensioning of type VARYING arrays at execution time). The dimensions may be arbitrary expressions containing variables defined in some covering scope. Such a dynamic array can be allocated from the stack by MAS or MMAS, operating at run-time.

Now consider an OWN array. The values of an OWN array must be preserved between an exit from its block through a subsequent re-entry into the block. This alone means that an OWN array cannot be allocated on the stack top; another data space, the heap, is needed for OWN variables and arrays.

If the OWN array has dimensions that are fixed at compile time, the compiler can simply allocate a group of words in the heap for the array. This obviously applies to simple OWN variables as well. Unfortunately, the Algol 60 rules permit redimensioning OWN arrays. The redimensioning cannot change the rank, but it can change the upper and lower limits. The change, of course, occurs if any of the dimensions is a variable expression. The Algol 60 rules furthermore specify that upon such a redimensioning, the value of every data element whose index set is valid under both the old and new dimensions must be preserved. That is, if the array element

A[15,7,20]

carries valid indices under both the old and the new dimensions, then its value must be preserved.

The new dimensions may be such that the overall array size is increased;

the new array will then not fit into the old allocated heap space. The old space must be released to a pool of available heap space, and new space must be allocated. All the compatible data values must also be copied into the new space.

If the new data space size is less than the old, then the old data space can be re-used, with some space possibly left over, available for other uses. Data values must in general be moved to new locations. The easiest way to move the values is to first copy them to a temporary heap or stack space, then copy them back into their correct positions. However, Ingerman [1961] solved the problem of moving the data values correctly in place, without requiring more than a few temporary memory cells.

The dynamic allocation of new heap space and the discarding of existing space calls for a memory management system for the heap in any practical implementation. We have regarded the heap and stack space as infinite, but of course in any implementation these spaces are limited by available memory and by fixed address word sizes. A limitation on stack space is not necessarily fatal to a program, since stack space tends to be reclaimed; no corpses of memory are left around. Heap space on the other hand will eventually consist of large dead areas separated by live areas. When a memory limit on heap space is reached, something should be done to make use of the dead areas, in an attempt to keep the program going, at least until all the available heap memory is filled with live data.

There is general agreement that the most efficient memory management system for this application is one in which new heap space is allocated from h downward, as needed. When a heap memory limit is reached, the program is interrupted, and the heap space is collapsed by collecting together all the live areas. This process is called *garbage collection*. The garbage collector must be able to determine the locations and sizes of each of the areas and whether an area is dead or live. It is sufficient to provide two pointers PREV'HEAP and BACK'PNTR and a flag LIVE in each data area, assuming that new areas are always allocated in the next heap area and are linked (with PREV'HEAP) in the order last-allocated-first. Then the first pointer value is also one word more than the end of the data area containing the pointer. (The purpose of the pointer BACK'PNTR will be clear in a moment.)

Given such a chain, the garbage collector would first build a list of live areas and their lengths on the stack by following the PREV'HEAP chain. It would then move the live areas to the heap bottom, fixing their links appropriately, in link order. Dead areas are thereby removed in this process.

The garbage collector must also update all references to the live areas. The update can be managed by requiring that exactly one pointer to a live heap area exist, on the stack. Let its location be P. We then need a pointer BACK'PNTR in each heap data area to P, so that the garbage collector can locate and adjust the pointer in P.

We see that each heap data area must therefore contain the following data structure—the remainder of the area is available for the data.

```
var PREV'HEAP, BACK'PNTR: integer,
    LIVE: Boolean;
```

Garbage collection can be implemented as a parallel process through a system of semaphores. See Gries [1977], Steele [1975], and Dijkstra [1976a] for details. Such an implementation would be very efficient if an independent processor could be dedicated to garbage collection. Those heap data areas not currently in use can be moved into dead areas while other processing continues concurrently.

9.7.4. Pascal Data Structures

We next consider the problem of allocation of Pascal data structures through AOC operations. This problem is a kind of generalization of the array access systems described in the previous section and in fact admits of two access soludions, one with direct index calculations and another with pointers.

These two kinds of structure organization are illustrated in Figs. 9.17 and 9.18. They correspond to the Pascal declarations

```
type NAMEPART=array [1. .20] of char;
type NAME=array [1. .3] of NAMEPART;
type VITA=record NAME: NAME;
                 AGE: integer;
                 SS: array [1. .3]
                    of integer;
                 SALARY: integer;
            end;
var LEONARD: VITA;
```

Figure 9.17 shows a matrix pointer implementation of LEONARD. At run-time, LEONARD is a pointer to a VITA list. VITA contains pointers to NAME, AGE, SS, and SALARY. Access to some primitive object at the end of a chain requires indirection, indexing, indirection, indexing, . . . , just as for array matrix pointers.

Figure 9.18 shows a packed implementation of LEONARD. Here the required data areas are packed into a contiguous group of data memory words, referenced by a textual address associated with LEONARD. The

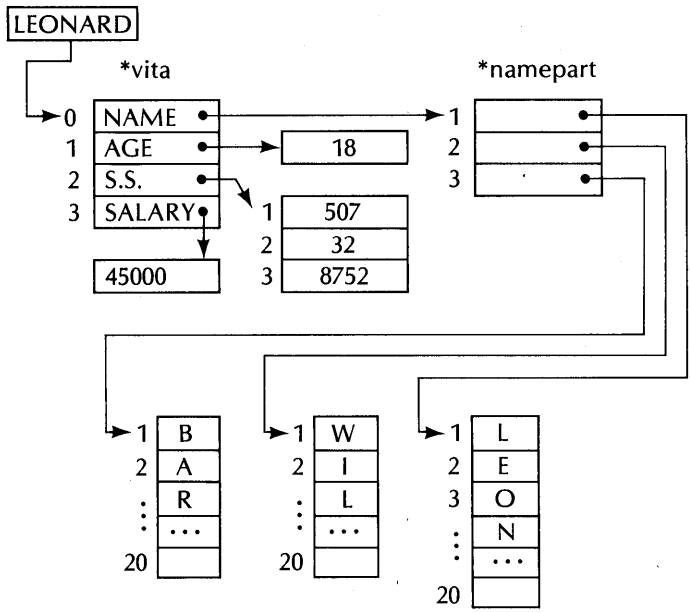


Figure 9.17. Run-time support structure for a Pascal VITA structure, using pointer matrices.

necessary offset of some data element is in general an expression whose operands are some combination of offsets within the structure and indices. Since the indices are run-time variables in general, the offset must be computed at run-time. Notice that a dope vector is unnecessary; each of the data items in figure 9.18 has a position known to the compiler, because the dynamic dimensioning of such structures is not supported in Pascal.

Let us begin with some observations that apply to Pascal data structures (and Pascal only!):

1. Every declared type has a fixed size.
2. Every type appearing in a structure bears some offset from its ancestor types, known at compile time. However, the offset depends on the ancestor. Recall that a given type may appear in more than one declaration and will therefore carry an offset that depends on the declaration details.
3. A type is allocated space and assigned an address only upon an appearance in a "var" or "constant" declaration. Until such an appearance, a Pascal type is an abstract description of a certain configuration of data.

We conclude from these observations that the compiler can assign a textual address T and allocate space to each declared variable name, e.g., ED in

var ED: <some type name>;

However, the address of each component datum in ED must be worked out on each reference. Only the variable ED can be assigned an address. The components of ED in the symbol table type structure will in general be shared by several different variables with different locations.

The mechanism of developing an address for a component of some variable is given by the following algorithm.

1. The *root name* is the name declared in a VAR or CONST declaration. It is assigned a transfer point [B,j] by the compiler. The next available transfer point is [B,j + j'], where j' is the number of words needed for the structure associated with the root name.
2. The size j' of a structure is determined by the compiler through a bottom-up tree walk through its structure lists, as follows. Each element in the structure carries a pair {S,D}; S is the size of the object and D is its offset relative to a related object.
   (a) For a primitive data object, {S,D} = {size of the object, 0}.
   (b) For an array object, the child will carry the pair {S,D}. If the upper

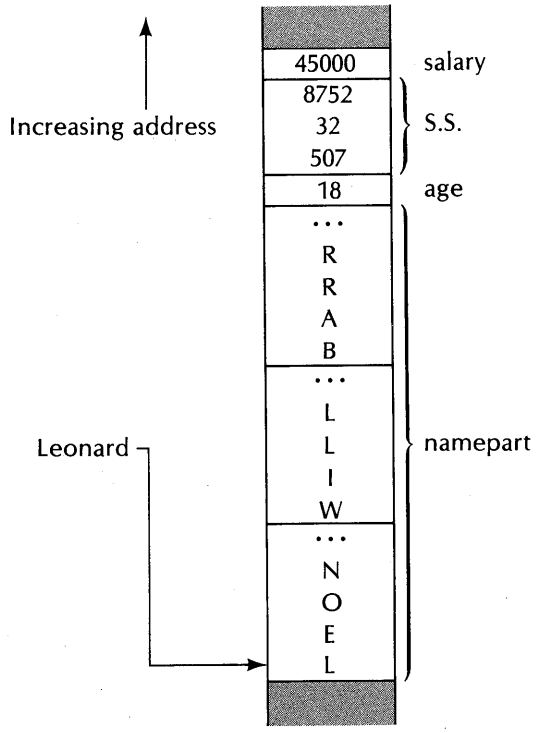


Figure 9.18. Run-time support structure for a VITA structure, using packed configuration.

and lower bounds of the array are U and L, then the array object will carry the pair {S*(U−L+1),D−L}.

(c) For a RECORD node, with selectors $s_1, s_2, \ldots, s_n$, and descendant types $T_1, T_2, \ldots, T_n$ (e.g. figure 8.6), let selector $s_i$carry the pair $\{S_i, D_i\}$. Then $D_1 = 0$, $D_2 = D_1 + S_1$, $D_3 = D_2 + S_2$, etc. In general, each of the D_i except the first will change from their original value of 0. Then S for the record node will be $D_n + S_n$, and D for the node will be 0.

(d) For a variant node (CASE), each of the case fields will carry some displacement S′, where S′ is the size of the CASE data object. The displacement D of the variant node will be 0, and its size will be S′ plus the maximum of the sizes of the field sizes. Thus the variant type structure is supported by an allocated memory space sufficiently large for the largest possible case.

For example, here are the TA's assigned to the components of LEONARD (cf. figure 8.8 and figure 9.18):

Component	Offset j in TA [B,j]
LEONARD.NAMEPART(0)	0 bytes
LEONARD.AGE	60
LEONARD.SS(0)	62 {an integer=2 bytes}
LEONARD.SALARY	68

The size of the LEONARD structure is 70 bytes.

9.7.5. Pascal Data Object Access

A structured data object is accessed by refining the base address [B,j] of its root node. Now each of the primitive data objects in the structure will carry some offset j′ from the root address. However, because of array indexing, the offset will in general be a run-time value. We propose a scheme for determining j′ for some data object described by a compound Pascal name. The scheme will emit code that locates the item, and if the code evaluates to a constant, then presumably the compiler will be able to identify this fact.

The emitted code might first be expressed as an AST of course, which permits various optimizations and reductions.

Given the base address [B,j] of a data object, we then consider the path downward through the type structure associated with the object (cf. section 8.3.3.3). Offset j is then altered as follows in passing through various nodes:

1. In passing through an array node with the index X (possibly an expression), lower dimension L, and pair {S,D},

$$j := j + S*(X-L) + D$$

Effectively, the array blocks of size S each begin at displacement D relative to their father node.

2. In passing through a record selector node associated with the pair {S,D},

$$j := j+D$$

3. In passing through a variant node, with pair {S,D},

$$j := j+D$$

When the data object at the end of the search path is reached, its transfer point is [B,j].

For example, LEONARD.SS(2) in figure 9.18, with a packed structure on byte boundaries and with 2-byte integers, will be accessed as follows:

- Let A′ = byte address of the LEONARD structure.
- Then A′+62 = byte address of the SS part of the structure (three 20-byte nameparts and one 2-byte age precedes this address)
- Then A′+62+2*(2−1) = A′+64 = byte address of LEONARD.SS(2).

Dynamic Pascal Objects

A pointer in Pascal is associated with some data structure. If pointer P is associated with type T, then P can only be set to point to a structure of type T. New memory space can be allocated to a pointer P, through the function *new*(P) This procedure allocates sufficient space (in the heap) for the type associated with P and also sets P to point to it. The procedure *new* is useful for building repeated data structures such as trees and lists.

We have already discussed the problem of dynamic allocation of memory space in the heap. We need only discuss the access of a data object through a pointer which is achieved essentially as follows:

```
LV   P     {contents of pointer, an address}
{evaluate offset of data object}
ADD
{address of data object is now on stack}
```

Exercises

1. Translate the following program to AOC code, using MAS. Give another translation using MMAS:

```
begin integer X;
   X:=READ;
   begin integer N;
      integer array A[0:X],
                    B[1:X,X+1:2*X];
      N:=READ;
      A[N+1]:=B[READ, READ+X]+25
   end
end
```

2. What modifications to AVA would provide run-time checking of array bounds? Consider two kinds of check: an overall check that the final addressed element is within the array somewhere, and a more particular check that each index is within its dimension.
3. What modifications to the matrix access system are needed for run-time checking of array bounds?
4. What modifications of AVA are needed for it to be called as any other AOC procedure, using the data stack, display, etc., of the calling procedure?
5. What modifications to MAS and MMAS are needed for these to operate on the user stack?
6. In figure 9.16, show that each of the six X values are correctly accessed through the access mechanism given.
7. Construct the stack configuration for the declaration

   var X: array [2:2, 3:4, −3:−2] of integer

   using MAS and MMAS. Let the dope vector reside at address 75 and stack top address s = 123 upon calling MAS or MMAS.
8. Compare the performances of AVA and matrix pointer access for an array of rank n. Assume that each memory fetch and store, that each unary or binary stack operation (other than multiply) requires 1 time unit, and that a multiply requires 5 time units.
9. Which requires more operations—MAS or MMAS? Does your answer depend on the rank or on the array dimensions?
10. Write a heap garbage collector in Algol, Pascal, or AOC code. Could this operate as a procedure on the user stack? When should it be invoked? Assume that the system imposes an absolute upper bound on

|h|. Are any new AOC instructions needed (i.e., in addition to those already introduced)?

11. Show the stack configuration resulting from the Pascal variable

```
var ADDR: ADDRESS
```

(cf. section 8.3.3.3), using packed accessing. Then give optimized AOC code to compute the addresses of each of its constituents.

9.8. Typed Procedures and Procedure Parameters

Programs are much clearer, more precise in meaning, and easier to write and maintain if the inputs and outputs of procedures are explicitly declared. An Algol procedure declaration has the general form

```
<type> PROCEDURE <name>
      ( <fp1> , <fp2> , . . . , <fpn> );
      <VALUE fp declarations> ;
      <specification fp declarations> ;
<procedure-body>
```

The $<fp_i>$'s are simple identifiers, called *formal parameters*, or FP's. Their attributes are declared in the <VALUE fp declarations> and in the <specification fp declarations>. The former list specifies those that are passed by VALUE; all others are passed by name. The latter list specifies the type (REAL, INTEGER, etc.) and whether an ARRAY or not.

The <procedure-body> is a block, consisting in general of a BEGIN-END pair enclosing local declarations, more procedure declarations, and a list of executable statements. Algol 60 also permits a single executable statement, without a BEGIN-END pair, as a <procedure-body>.

The procedure <type> must appear if the procedure is to be used as a function; this is the type of the returned value, REAL, INTEGER, etc. The value of a typed procedure is assigned-to within the <procedure-body> by a statement of the form

```
<procedure name> := <expression>
```

The parameters e_1, e_2, etc. that appear in a call

```
<procedure name> ( e1, e2, . . . )
```

are called *actual parameters* or AP's. A procedure call is handled roughly as follows in the AOC machine:

1. If the procedure is typed, a word is allocated on the stack for its return value. Upon a procedure exit, this word will be left in place, exactly as needed for whatever expression evaluation was taking place at the moment of the call.
2. For each of the actual parameters, the compiler considers the relation of the actual parameter to the corresponding formal parameter declarations. Algol 60 permits passing parameters by *value* or by *name*. Other parameter passing mechanisms have been proposed—we will consider passing by *reference* later. The compiler must emit code that will cause a value, an address, or something to be written on the stack for each actual parameter.
3. Finally, the procedure is called through the CP instruction. The next execution step is then the first line of the <procedure-body> code. Within the procedure, the material previously written on the stack as actual parameters is now viewed as associated with the formal parameters. They are accessed through textual addresses of the form [B,j], where j is negative. Upon an exit (RTN) from the procedure, this material must be removed, leaving at most the procedure return value on the stack. Adjustments to the display are also made, and the next executed statement is the statement following the procedure call in the calling instruction sequence.

Certain abstract rules govern the semantics of procedures and procedure calls. These are the *procedure copy rule* (PCR), rules for *call by value* (CBV), for *call by name* (CBN), and a *typed procedure exit rule* (TPER). These rules specify the semantic effect of calls.

9.8.1. The Procedure Copy Rule PCR and Call by Name CBN

The execution of a (typed) procedure call

$$P(e_1, e_2, \ldots, e_n)$$

is effectively (but not in practice) carried out by:

1. Replacing the call by a copy of the body of the procedure declaration, using TRR to avoid variable conflicts.
2. Replacing all occurrences (in the copy) of the formal parameters $x_1, x_2, \ldots, x_n$ by the corresponding actual parameters enclosed in parentheses, i.e., (e_1), (e_2), . . . Again, TRR is invoked to avoid variable name conflicts.

These steps are taken before the actual parameters are evaluated. For example, a simulation of the program

```
begin  integer A, B;
    procedure Q(X,Y); integer X,Y;
    begin  X:=Y*Y
    end;
    B:=3;
    Q(A,B+6)
end
```

would lead to the program

```
begin  integer A=?, B=3;
    procedure Q(X,Y); integer X,Y;
    begin  X:=Y*Y
    end;
    B:=3;
    begin A:=(B+6)*(B+6) end
end
```

Thus the computation of B+6 would be delayed until the formal parameter appears; it would also be done for each appearance, twice in this example. The value of A is also changed by the procedure call.

It should be obvious that a compiler cannot implement CBN by substituting strings in the source—the specific substitution depends on which of several possible procedure calls are being executed. Furthermore, the source has disappeared when the program is executed. We shall describe an equivalent run-time system for implementing CBN in section 9.8.7.

9.8.2. Call by Value (CBV)

Call by value, or CBV, is defined by the following rule:

Value Parameter Evaluation Rule (VPER). A procedure declaration such as

```
procedure P(A,B,C); value A,C;
      integer A,B,C;
  begin . . . end
```

is equivalent to

```
procedure P(A',B',C');
    integer A', B', C';
  begin integer A,C;
      A:=A';
      C:=C';
      begin . . . end
  end
```

where A′ and C′ are new variables not appearing in "begin . . . end".

The VPER is just a formal way of stating that CBV parameters are evaluated at the time of the call and stored in new locations local to the procedures. Only the local copies are affected by the procedure instructions. An assignment to a CBV FP will change the local copy but not the AP outside the assignment.

The following example illustrates VPER and TPER:

```
begin integer A;
   integer procedure S(X); value X;
        integer X;
   begin S:=X+1 end;
   A:=3;
   PRINT(S(A*2))
end
```

and is equivalent, by VPER and TPER, to

```
begin integer A;
   integer procedure S(X′); integer X′;
   begin integer X;
      X:=X′;
      begin S:=X+1 end
   end;
   A:=3;
   PRINT(S(A*2))
end
```

Through simulation by copying, we arrive at the next program state:

```
begin integer A=3;
   integer procedure S(X′); integer X′;
   begin integer X;
      X:=X′;
      begin S:=X+1 end
   end;
   A:=3;
   PRINT(begin integer X=6;
            X:=(A*2);
            begin S:=X+1 end
         end)
end
```

Of course, the "begin . . . end" material within PRINT is syntactic nonsense

in Algol 60, but the spirit of the operations is clear. The application of TPER then yields

```
begin integer A=3;
    integer procedure S(X'); integer X';
    begin integer X;
        X:=X';
        begin S:=X+1 end
    end;
    A:=3;
    PRINT(7)
end
```

9.8.3. More About Free Variables

Recall that we found it important to apply TRR before TPCR in implementing procedure calls, in order to resolve the free variable problem. We can illustrate another facet of the free variable problem with the following example:

```
begin procedure G(X); integer X;
    begin integer I;
        I:=X-1;
        if X=0 then PRINT(X)
               else G(I)
    end;
    G(2)
end
```

Notice first that no applications of TRR are necessary, since the variables are all distinct. In simulating this program with TPCR, we arrive at the state:

```
begin procedure G(X); integer X;
    begin integer I;
        I:=X-1;
        if X=0 then PRINT(X)
               else G(I)
    end;
    begin integer I=1;
        I:=2-1;
        if I=0 then PRINT(2)
→              else G(I)
```

If we apply TPCR again, but neglect TRR, we arrive at

```
begin procedure G(X); integer X;
    begin integer I;
      I:=X-1;
      if X=0 then PRINT(X)
             else G(I)
    end;
    begin integer I=1;
      I:=2-1;
      if I=0 then PRINT(2)
             else begin integer I=?;
→                 I:=I-1;
                    if I=0 . . .
```

Whereas, using TRR to make the I's distinct, we get

```
begin procedure G(X); integer X;
    begin integer I;
      I:=X-1;
      if X=0 then PRINT(X)
             else G(I)
    end;
    begin integer I=1;
      I:=2-1;
      if I=0 then PRINT(2)
             else begin integer I1=?;
→                 I1:=I-1;
                    if I1=0 then PRINT(I)
                           else G(I1)
```

The outcome clearly depends on applying TRR before TPCR. In the former case, the outcome is undefined since I is undefined, whereas in the latter case, "1" is printed.

9.8.4. Fortran Parameter-Passing Rules

The Fortran parameter-passing rules are neither CBV nor CBN, rather something in-between, called *call by reference*, CBR. A reference (memory address) is always passed. The reference will be a pointer to a variable, to an indexed variable (pointer to the variable offset by the index value), to a constant, or to some temporary assigned to the value of an actual parameter expression. An actual parameter expression is first evaluated, then assigned to a temporary memory location; the passed reference is to the temporary.

For the purposes of parameter passing, a variable or indexed variable is not considered an expression.

We see that the rule for actual parameter expressions is CBV, while the rule for other parameters is CBN. The constant rule is neither and depends on the implementation in general. In some implementations, a temporary copy of the constant is made and an address of that copy is passed. In others, the address of the constant itself is passed. In the latter implementations, the following program illustrates a run-time bug that yields a surprising and dangerously unpredictable result:

```
SUBROUTINE MIKE(X)
X=6
END

MIKE(7)
PRINT(7)
```

Here, if only one "constant" 7 exists in memory, and its address is passed to MIKE, the constant 7 is turned into a 6, and "6" is printed. However, there might be several copies of 7, or a temporary might be assigned to 7, in which case "7" would be printed. A Fortran compiler can protect a programmer to some extent by making a temporary copy of the constant, then passing the address of the copy. At least then the constant as seen outside the procedure will not be changed by the procedure. However, something is wrong with the programmer's concept of the procedure's application, and the compiler cannot bring this to his attention.

9.8.5. Implementation of CBV and Typed Procedures

CBV is much easier to implement than CBN, so we shall consider it first. Although CBV is defined in terms of CBN, we shall develop a direct implementation that does not depend on CBN.

No new operations are required. We only need to modify RTN. The general procedure call rule is simple: when an actual parameter is to be stacked, evaluate it and leave its value behind on the stack. Within the called procedure, the value is accessed by a negative offset (known to the compiler) relative to the base of the stack marker (D(B)). Since the compiler will not otherwise know the location of the stack marker base, we require that a BE always be executed upon a procedure entry.

The stack marker then always contains five words as follows:

static pointer	D(b)	(from CP)
display top	b	(from CP)
return address	i	(from CP)
working pointer	S+n+2	(from BE n)
static pointer	D(b)	(from BE n)

(*Note:* PE changes b and comes between CP and BE. Hence the static pointers are different in general. Thus sm = 5 in the following discussion.)

The return command RTN must remove the formal parameter material as well as the stack marker, so we redefine the command as follows:

```
RTN  m:  Return from procedure
           m = formal parameter words
         Go(s)          {reset the display}
         s ← s − m − sm {reset s; sm = marker size}
```

The actual parameters are evaluated from left to right, so that in the call

$$P(e_1, e_2, e_3)$$

the value of e_1 is deepest in the stack. This call yields the display and stack configuration shown in figure 9.19.

The textual addresses (TA's) required in the procedure body, at block level N, for the FP's corresponding to e_1, e_2, and e_3 are

for e_1: [N, −3 − sm]
for e_2: [N, −2 − sm]
for e_3: [N, −1 − sm]

In general, the TA for x_i in the procedure with the declaration

procedure $P(x_1, x_2, \ldots, x_i, \ldots x_m)$; . . .

is [N,i − m − sm]. Note that (i − m − sm) can be assigned by the compiler as soon as the ")" of the formal parameter names is scanned. In practice, different kinds of parameters may require different numbers of bytes each, so

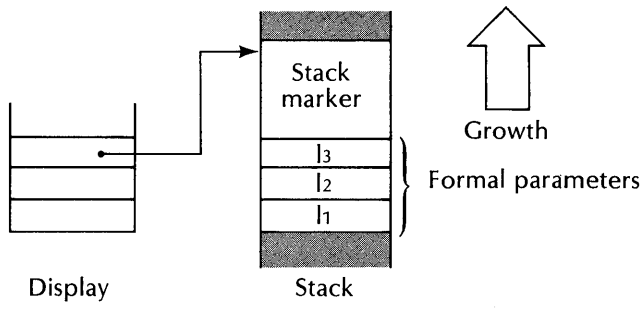


Figure 9.19. Stack configuration upon procedure call $P(e_1, e_2, e_3)$.

that the formal parameter declaration must be scanned before addresses can be assigned to the formal parameters.

9.8.6. Typed Procedure Return Value

A typed procedure can be called as a stand-alone statement, or as part of an expression, e.g.

```
P(5,Y);      {statement form}
A:=15+P(5,Y);    {expression form}
```

In either case, the compiler reserves one word on the stack, perhaps via LC 0, before loading any parameters. Inside the procedure, any assignment to its name is treated as an assignment to a variable with the TA [N, −m−sm−1]. Upon an exit, the return value will be the only remaining object on the stack. Furthermore, nothing in the stack deeper than the return value will be affected by the procedure, except through specific up-level address references in the procedure.

CBV example

Consider the following program, introduced previously:

```
begin integer A;
   integer procedure S(X); value X; integer X;
   begin S:=X+1 end;
   A:=3;
   PRINT(S(A*2))
end
```

This program will translate to

```
        BE      1       begin integer A;
        JP      L       {to get around the procedure code}
S       PE      0       integer procedure S(X); . . .
        BE      0       begin
        LV      1,−6    S:=X+1    {sm=5}
        LC      1
        ADD
        STD     1,−7    {return value}
        EB              end
        RTN     1
L       LC      3       A:=3
        STD     0,0
```

```
LC          0       {reserve return location}
LV          0,0     A*2
LC          2
MULT
CP          S       S(A*2)
PRINT               PRINT(S(A*2))
EB
HALT
```

9.8.7. Implementation of CBN Procedure Parameters

A CBN parameter will be implemented by creating a procedure that can evaluate the actual parameter, then passing a reference to the procedure. Such a procedure is called a *thunk*, a name devised by P. Z. Ingerman.

For example,

```
begin integer procedure SUM(F,N);
        value N; integer N;
        integer procedure F;
      begin integer I,S;
        I:=1;  S:=0;
   LOOP:
        if I<N then
        begin S:=S+F(I);
              I:=I+1;
              go to LOOP
        end;
        SUM:=S
      end;
      integer procedure SQ(X); value X;
          integer X;
      begin SQ:=X*X end;
      PRINT(SUM(SQ,READ))
end
```

will print the sum of the squares of the integers returned by READ.

$$\sum_{i=1}^{\text{READ}} i^2$$

To implement CBN procedure parameters we need two new operations:

```
LL  l:   Load label l as a TP
         s ← s+1
         C(s) ← D(b),b,l {recall the
              packing notation}

CPV B,j: Call procedure as variable
         s ← s+1
         C(s) ← D(b),b,i    {save TP}
         Go(C(D(B)+j))          {then transfer}
```

Notice that CP can be regarded as a degenerate case of CPV in which the display need not be restored because the call is within the block where the procedure was declared, guaranteeing that the procedure's context is already in the display.

Also notice that there are two transfer points involved here: the one being created for eventual use by RTN and the one being transferred to.

We pass a procedure name by loading its transfer point. It is called by CPV instead of CP.

The translation of the summation program given above is then

```
        BE      0       begin
        JP      L1      {get around procedure code}
SUM     PE      0       integer procedure SUM(F,N); .
        BE      2       begin integer I,S;
        LC      1       I:=1
        STD     1,0
        LC      0       S:=0
        STD     1,1
LOOP    LE      1       LOOP:
        LV      1,0     if I<N then
        LV      1,-6
        LS
        NOT
        JIF     L2
        BE      0       begin
        LV      1,1     S
        LC      0       {return value place}
        LV      1,0     I
        CPV     1,-7    F(I)
        ADD
        STD     1,1     S:=S+F(I)
        LV      1,0     I:=I+1
        LC      1
```

```
        ADD
        STD     1,0
        JP      LOOP        go to LOOP
        EB
L2      LV      1,1         SUM:=S
        STD     1,-8
        EB                  end
        RTN     2
SQ      PE      0           integer procedure SQ(X); . . .
        BE      0           begin
        LV      1,-6        SQ:=X*X;
        LV      1,-6
        MULT
        STD     1,-7
        RTN     1
        EB                  end
L1      LC      0           {return value for SUM}
        LL      SQ          {procedure label}
        READ
        CP      SUM
        PRINT               PRINT(SUM(SQ,READ))
        EB
        HALT
```

9.8.8. Implementation of CBN Label Parameters

Algol 60 permits a statement label to be passed to a procedure. Such a label can be used as an error exit, permitting the program to branch back through several levels of procedure calls to some common point. By modern standards of structured programming, this is not a desirable language feature, as it means that no procedure call containing a label can be counted upon returning to the statement following the procedure. However, the language does require this feature, so let us see how a passed label can be implemented.

Here is an example:

```
begin integer procedure RECIP(X,E);
        value X; integer X; label E;
    begin if X=0 then go to E;
          RECIP:=1/X
    end;
    PRINT(RECIP(READ,ER) +
      RECIP(READ,ER))
  ER:
end
```

In this program, if either number read is zero the computation of the sum of the reciprocals is cut short by the transfer to ER.

The implementation of label parameters goes much as procedure parameters. We shall use LL and invent a new instruction

JPV B,j: Jump to variable at address B,j
Go(C(D(B)+j))

Once again, JP can be viewed as a degenerate case of JPV. Recall that procedure Go reorganizes the display to correspond to the previous environment, that associated with the branch target.

The translation of the above program then is

```
        BE      0       begin
        JP      L1
RECIP   PE      0       integer procedure RECIP . . .
        BE      0       begin
        LV      1,-7    if X=0 then
        LC      0
        EQ
        JIF     L2
        JPV     1,-6    go to E
L2      LC      1       RECIP:=1/X
        LV      1,-7
        DIV
        STD     1,-8
        EB              end
        RTN     2
L1      LC      0       {return value for RECIP}
        READ
        LL      ER
        CP      RECIP
        LC      0
        READ
        LL      ER
        CP      RECIP
        ADD
        PRINT           PRINT(RECIP(READ,ER) + RECIP(READ,ER))
ER      LE      0
        EB
        HALT
```

9.8.9. Summary of Procedure Parameter Mechanisms

The possible relationships of actual to formal parameters for Algol-like languages are summarized in figure 9.20. Algol 60 supports only CBV and CBN, while Fortran supports CBR. We have shown all three cases in the figure for the sake of generality.

The actual parameter may be

1. A constant.
2. A variable name or indexed variable name.
3. An expression.
4. A procedure name.

By *expression*, we mean any expression other than a constant, a variable name, or an indexed variable name, which fall under the first two categories. By *procedure name*, we mean that the name of a procedure is passed, to be called within the callee procedure, not a procedure intended to be called before passing the parameter. A passed procedure name will never carry actual parameters when it appears as an actual parameter.

The uses of the formal parameter may be classified as follows:

<table>
<tr><td></td><td colspan="4">ACTUAL PARAMETER</td></tr>
<tr><td></td><td>F.P.</td><td>Constant</td><td>Variable</td><td>Expression</td><td>Procedure name</td></tr>
<tr><td rowspan="3">Variable assigned-to</td><td>CBV</td><td colspan="3">Affects local copy</td><td rowspan="3">ERROR</td></tr>
<tr><td>CBR</td><td>ERROR*</td><td rowspan="2">Affects actual parameter</td><td>ERROR*</td></tr>
<tr><td>CBN</td><td>ERROR*</td><td>ERROR*</td></tr>
<tr><td rowspan="3">Variable fetched only</td><td>CBV</td><td rowspan="3">Returns constant value</td><td>Variable evaluated before call</td><td>Expression evaluated before call</td><td rowspan="3">ERROR</td></tr>
<tr><td>CBR</td><td rowspan="2">Variable value at fetch</td><td rowspan="2">Expression evaluated at fetch</td></tr>
<tr><td>CBN</td></tr>
<tr><td>Procedure call</td><td>(any)</td><td colspan="3">ERROR</td><td>Procedure called</td></tr>
</table>

*Difficult error to detect in general

Figure 9.20. Procedure parameter passing cases, with call-by-value (CBV), call-by-reference (CBR) and call-by-name (CBN).

1. As a variable that is assigned-to somewhere.
2. As a variable that is never assigned-to, but may be fetched.
3. As a procedure name to be called.
4. As an actual parameter to be passed to another procedure.

We shall deal with case 4 separately; in principle, this can always be resolved to one of the other three if the called procedure is considered. Recursive calls that only pass parameters around among each other are barred from consideration.

Now the declarations for a formal parameter in Algol 60 provide the following attributes:

1. Whether CBV or CBN (recall that CBR is not in Algol 60),
2. Whether a procedure or not.

We can assume that the formal parameter attributes are known upon analyzing any call. Note that "assigned-to" is not among the attributes; it can only be determined through an analysis of the procedure statements.

For the sake of generality, we add the attribute REFERENCE to those already in Algol 60, to indicate that the formal parameter is CBR. We can then distinguish CBV, CBN, and CBR from the FP declarations.

Now consider passed procedure names (figure 9.20). A PROCEDURE formal parameter can only be associated with a procedure name actual parameter; any other actual parameter is a compile-time error and is easy to detect. Also, any use of a PROCEDURE formal parameter other than as a procedure call is illegal.

Next consider a constant actual parameter. A constant should not be assigned-to, but may be fetched. In principle, the compiler can detect such errors, but an elaborate mechanism is needed, as we shall see. For CBV, "assigning-to" a constant merely means that the local copy of the constant is altered; the actual parameter is unaffected. Such a use of a CBV formal parameter is valuable, as a means of conserving local variable space. For example, procedures MAS, SETP and MMAS (above) use this characteristic.

An expression's actual parameter also should not be assigned-to, but can be fetched. Here, CBR and CBN are fundamentally different. In CBR, the expression is evaluated just before the call, at the moment at which the actual parameter is processed by the compiler. In CBN, the expression is not evaluated before the call, but is evaluated afresh upon each appearance of its corresponding formal parameter within the called procedure. Clearly, a thunk must be created by the compiler upon processing the actual parameter for CBN. The thunk label is passed as the actual parameter, and the thunk is called when the formal parameter reference is processed in the procedure. The thunk will then evaluate the expression and return its value, as needed.

The CBN and CBR expression evaluation result can clearly be different. For example, if the expression contains free variables, their value may be changed by the callee between formal parameter references. Then CBN will return different results on different references, whereas CBR will always return the same result.

Of course, CBR can return different results on different references, but only because a specific assignment to that formal parameter was made, or another CBR formal parameter carries the same address and was assigned-to. The latter situation is called *aliasing*—two apparently different formal parameters refer to the same data area. Aliasing is a fertile source of program errors.

If the actual parameter is a simple gr indexed variable, then an assignment to it within the callee procedure makes sense and is a useful operation. However, again, CBR and CBN can yield different results, especially for an indexed variable. The index is an expression in general, and in CBN the index is evaluated on each reference. In CBR the index is evaluated before the call, and the appropriate indexed variable address is passed; the index is thereupon effectively fixed while the callee is active.

The following (nonsensical) program illustrates all the cases of figure 9.20.

```
begin
   integer I,J,K;
   integer array IA[0:15];
   procedure Q(A,B,C,D);
      value A;          {A is CBV}
      reference B;      {B is CBR}
      integer A,B,C;    {C is CBN}
      procedure D; {D is a procedure name}
   begin
      {variables assigned-to}

      A:=10;     {OK in all cases}
      B:=10;     {OK except for
                  constant or expression AP}
      C:=10;     {OK except for
                 constant or expression AP}
      D:=10;     {ERROR}

      {variables fetched}

      I:=A;      {OK in all cases}
      I:=B;      {OK in all cases}
      I:=C;      {OK in all cases}
      I:=D;      {ERROR}
```

```
{variables called}
   A;          {ERROR}
   B;          {ERROR}
   C;          {ERROR}
   D(I,I,I,Q)  {OK, assuming the parameters
                match the called procedure}
end;
{now we give some typical calls}
Q(5,5,5,5);    {constants passed—3 errors}
    **↑   {*=hard to detect, ↑=easy to detect}
Q(I,I,I,I);    {simple variables—one error}
       ↑
Q(IA[I+J], IA[I+J], IA[I+J], IA[I+J]);
                                ↑   {1 error}
Q(I+J, I+J, I+J, I+J);    {3 errors}
        *    *    ↑
Q(Q,Q,Q,Q);    {procedure names—3 errors}
  ↑ ↑ ↑
end
```

The errors marked * in figure 9.20 and in the above program are difficult to detect in a one-pass compiler, but possible to detect in a two or more pass compiler. These depend on detecting an assigned-to condition of a CBN or CBR formal parameter, when the actual parameter is a constant or expression. In order to detect such an error upon processing a call, the attribute "assigned-to" or "not assigned-to" must be associated with each CBN and CBR formal parameter. The "assigned-to" attribute is set if either of the following conditions is met:

1. The formal parameter is set somewhere within the procedure, i.e., as the left member of an assignment, the target of a READ, etc.
2. The formal parameter is passed by reference or name to another procedure, such that the corresponding formal parameter is "assigned-to."

The first condition requires one pass to establish "assigned-to" for its formal parameters. More passes are needed to check out the second condition if the first is not met. For example, the following program requires three passes to fully establish the "assigned-to" status for each of the formal parameters:

```
begin
   integer I;
   procedure Q1(A1);
      integer A1;  {A1: CBN}
   begin
      Q2(A1);
   end;

   procedure Q2(A2);
      integer A2;
   begin
      Q3(A2);
   end;

   procedure Q3(A3);
      integer A3;
   begin
      A3:=15
   end;

   Q1(14);
   Q1(I+J);    {each of these calls
                contains an error}
end
```

Note that the FP is CBN in each procedure. On the first pass, A3 only is "assigned-to." On the second pass, because A2 is passed to Q3 by CBN, A2 is "assigned-to." On the third pass, A1 is found to be "assigned-to." Obviously, programs can be devised that will require an arbitrary number of passes to fix the "assigned-to" status of all FP's.

A more practical strategy is to not attempt a compile-time check of these errors, in the hope that they will be manifest at run-time. This is not in the best interests of the user, but certainly simplifies the compiler.

9.8.10. Call by Name Implementation

Figure 9.20 illustrates a special problem with CBN—an efficient evaluation approach seems to depend on the nature of the actual parameter. If the AP is a constant, it may simply be loaded on the stack and fetched by a LD. If it is a simple variable, the variable's address may be placed on the stack and evaluated by the pair of instructions LA, CONT. However, an indexed variable and an expression require a thunk—they are to evaluated at the point of fetch (or store, in the case of an indexed variable).

Now we cannot afford to provide three different mechanisms within a procedure to deal with the three different AP cases. We need one common

method to deal with all three in a uniform manner, and that has to be a thunk in every case.

If a thunk is needed for an AP, the procedure caller will construct it—the caller after all knows the character of the AP. In every case, the thunk should return the address of something, since an address is needed for a variable that is assigned-to. Then the callee need only call the thunk associated with the FP and use the returned address in a standard way to either fetch the value or store into a variable.

The thunk for a simple variable should return the address of the variable. The thunk for an indexed variable should evaluate the index and return the address of the indexed variable. The thunk for a constant should place the constant in some addressable location (we discuss this later), and return the address of that location. Finally, the thunk for an expression should evaluate the expression, store the result in an addressable location, and return the address of that location.

Now what location is appropriate for a constant or expression temporary value in a thunk? If a temporary location is allocated by the thunk during its execution, its address could be returned; however, the address will point to a location above the stack top upon return, an illegal address in general. Clearly, a temporary is needed just before the thunk call. We therefore must adopt the thunk calling convention:

```
LC  0   {word to receive the returned address}
LC  0   {word to receive a temporary value, if any}
CPV . . .   {thunk call}
```

Obviously, the two LC's can be combined into an INCS 2. Then if the address is wanted from the thunk, the instruction

```
DECS 1
```

following the thunk call is appropriate.

If the value is wanted, we need a special instruction RAV as follows:

```
RAV:  Replace address by value
      C(s−1) ← C(C(s−1))
      s ← s−1
```

RAV uses the returned address in location s − 1 to fetch the value, replaces the address by the value, then drops the stack top. This peculiar instruction should only be used after a thunk call to obtain a value.

Now some typical thunks might be written as follows, assuming the thunk is written at block level 3 (the block level it is written in is immaterial, except that during execution, its environment must be that of the caller setting it up):

```
{thunk for a constant c}
PE      3
BE      0       {no local variables, but b=4 now}
LC      c       {load the constant}
STD     4,-6    {value is just below stack marker}
LA      4,-6
STD     4,-7    {address is below value field}
EB
RTN     0

{thunk for an expression A*B}
PE      3
BE      0
LV      A       {whatever A's textual address is}
LV      B
MULT
STD     4,-6
LA      4,-6
STD     4,-7
EB
RTN     0

{thunk for an indexed variable X[I+J]}
PE      3
BE      0
LA      X
LV      I
LV      J
ADD
ADD
STD     4,-7
LV      4,-7
CONT
STD     4,-6
EB
RTN     0
```

Since some of these instructions will always appear in a thunk definition, they might be combined into single instructions, for the sake of efficiency. For example, the combination

```
STD     4,-6
LA      4,-6
STD     4,-7
```

needs only the thunk block level (4). Also, the entry pair PE, BE and the exit pair EB, RDN are standard for a thunk. Special thunk call instructions that return either address or value could also be devised that combine some functions of the above call sequences.

Example

The following example illustrates CBN and CBV for several kinds of actual parameter and generation of the necessary thunks for CBN. Since CBR is an obvious variation on CBV, we will not illustrate CBR.

```
begin integer A,B;
   integer procedure INC(X,CHANGE);
      value CHANGE; integer X;
      boolean CHANGE;
   begin INC:=X+1;
      if CHANGE then X:=X+1
   end;
   A:=INC(0,FALSE);  {constant by name}
   B:=INC(A,TRUE);   {variable by name}
   B:=INC(A*B,FALSE); {expression by name}
   begin integer procedure INCA;
      begin INCA:=INC(A,TRUE) end;
      A:=INC(INCA,FALSE)
         {procedure name passed}
   end
end
```

The textual addresses of A,B,X and CHANGE are:

```
A: [0,0]
B: [0,1]
X: [1,-7]
CHANGE: [1,-6]
```

The TA of the return value of INC will then be [1,−8].

The translation of the previous program is given next.

```
        BE      2     begin integer A,B;
        JP      L1    {branch around procedure code}
INC     PE      0     integer procedure INC(X,CHANGE); . . .
        BE      0
        INCS    2     {locations for thunk address, value}
```

```
        CPV   1,-7 {call thunk for X}
        DECS  1    {leave address behind}
        LC    1
        ADD
        STD   1,-8 INC:=X+1
        LV    1,-6
        JIF   L2   if CHANGE then
        INCS  2    {thunk call for X again}
        CPV   1,-7
        DECS  1
        INCS  2
        CPV   1,-7 {another thunk call for X}
        RAV        {value wanted this time}
        LC    1
        ADD
        ST         X:=X+1
L2      EB         end
        RTN   2
L1      LC    0    {return value cell}
        LL    TH1
        JP    L3
TH1     PE    0    {start of thunk for constant 0}
        BE    0
        LC    0    {the constant}
        STD   1,-6
        LA    1,-6
        STD   1,-7
        EB
        RTN   0
L3      LC    FALSE
        CP    INC
        STD   0,0  A:=INC(0,FALSE)
        LC    0
        LL    TH2
        JP    L5
TH2     PE    0    {start of thunk for variable A}
        BE    0
        LA    0,0
        STD   1,-7
        LV    1,-7
        CONT
        STD   1,-7 {value may or may not be wanted}
        EB
        RTN   0
```

```
L5      LC      TRUE
        CP      INC
        STD     0,1     B:=INC(A,TRUE)
        LC      0
        LL      TH3
        JP      L8
TH3     PE      0       {start of thunk for A*B}
        BE      0
        LV      0,0
        LV      0,1
        MULT
        STD     1,-6
        LA      1,-6
        STD     1,-7
        EB
        RTN     0
L8      LC      FALSE
        CP      INC
        STD     0,1     B:=INC(A*B,FALSE)
        BE      0       begin
        JP      L10
INCA    PE      1       integer procedure INCA;
        BE      0
        LC      0
        LL      TH2     {label of thunk for A}
        LC      TRUE
        CP      INC
        STD     2,-7    INCA:=INC(A,TRUE)
        EB              end
        RTN     0
L10     LC      0
        LL      TH4
        JP      L11
TH4     PE      1       {start of thunk for INCA}
        BE      0
        LC      0
        CP      INCA
        STD     2,-7
        LA      2,-6
        STD     2,-7
        EB
        RTN     0       {end of INCA thunk}
L11     LC      FALSE
        CP      INC
```

```
STD    0,0    A:=INC(INCA,FALSE)
EB            end
EB            end
HALT
```

9.8.11. CBV versus CBN

In the previous section, we contended that the callee (the called procedure) should be ignorant of the nature of the AP's in any particular call. By the same token, it seems desirable that the caller should also be ignorant of whether its parameters are passed by value or by name. If the caller does know through a required declaration of the procedure, well and good, but Algol 60 doesn't require such a local declaration.

In order to achieve this kind of complete isolation of caller and callee—neither knowing the conventions of the other—the caller must assume the worst and always pass a thunk label. The callee must always expect a thunk label. This label is obviously inefficient in stack space and execution time, but is the price that must be paid for such isolation.

Because this general situation is so clumsy in implementation, most modern languages do not support CBN.

Either CBR or CBV may be dropped to support quite general procedure calls. In Fortran, only CBR is supported. If only CBV is supported, a pointer may be passed by value, effectively supporting CBR.

9.9. Summary of AOC Instructions

1. Stack manipulation instructions

LC c: Load constant
$s \leftarrow s+1$
$C(s) \leftarrow c$

LA B,j: Load address
$s \leftarrow s+1$
$C(s) \leftarrow D(B)+j$

LV B,j: Load value
$s \leftarrow s+1$
$C(s) \leftarrow C(D(B)+j)$

LL l: Load label
$s \leftarrow s+1$
$C(s) \leftarrow D(b),b,l$

STD B,j: Store direct
$C(D(B)+j) \leftarrow C(s)$
$s \leftarrow s-1$

ST: Store
$C(C(s-1)) \leftarrow C(s)$
$s \leftarrow s-2$

RAV: Replace address by value
$C(s-1) \leftarrow C(C(s-1))$
$s \leftarrow s-1$

CONT: Contents
$C(s) \leftarrow C(C(s))$

INCS: Increment stack pointer $(C(s) \geq 0)$
$C(s+C(s)) \leftarrow s$;
$s \leftarrow s+C(s)$

INCS n: Increment stack pointer $(n \geq 0)$
$s \leftarrow s+n+1$

DECS: Decrement stack $(C(s) \geq 0)$
$s \leftarrow s - C(s) - 1$

DECS n: Decrement stack $(n \geq 0)$
$s \leftarrow s-n$

2. Expression computing instructions

ADD: $C(s-1) \leftarrow C(s-1) + C(s)$
$s \leftarrow s-1$

SUB: $C(s-1) \leftarrow C(s-1) - C(s)$
$s \leftarrow s-1$

MULT: $C(s-1) \leftarrow C(s-1) * C(s)$
$s \leftarrow s-1$

DIV: $C(s-1) \leftarrow C(s-1) / C(s)$
$s \leftarrow s-1$

NEG: $C(s) \leftarrow - C(s)$

EQ: $C(s-1) \leftarrow$ if $C(s-1) = C(s)$ then
TRUE else FALSE
$s \leftarrow s-1$

LS: $C(s-1) \leftarrow$ if $C(s-1) < C(s)$ then
TRUE else FALSE
$s \leftarrow s-1$

```
GT:  C(s-1) ← if C(s-1) > C(s) then
             TRUE else FALSE
     s ← s-1

NOT: C(s-1) ← if C(s) = FALSE then
             TRUE else FALSE

AND: C(s-1) ← if C(s-1)=TRUE and
                C(s)=FALSE then
                TRUE else FALSE
     s ← s-1

OR:  C(s-1) ← if C(s-1)=TRUE or
                C(s)=FALSE then
                TRUE else FALSE
     s ← s-1
```

3. Block instructions

```
BE n: Block entry
      C(s+1) ← s+n+2        {save WP}
      C(s+2) ← D(b)         {save SP}
      b ← b+1
      D(b) ← s+3
      s ← s+n+2

EB:   Exit block
      s ← D(b)-3
      b ← b-1
```

4. Jump instructions

```
JP l:   Jump
        i ← l

JIF l:  Jump if false
          if C(s)=FALSE then i ← l
          s ← s-1

JPV B,j: Jump through variable
           Go(C(D(B)+j))
             {See section 9.6.4}

LE B:   Label entry
          b ← B
          s ← C(D(b)-2)
```

5. Procedure instructions

CP l: Call procedure
s ← s+1
C(s) ← D(b),b,i
i ← l

PE B: Procedure entry
b ← B

RTN m: Return
Go(C(s)) {see section 9.6.4}
s ← s−m−sm

CPV B,j: Call procedure through variable
s ← s+1
C(s) ← D(b),b,i
Go(C(D(B)+j))

6. Input-output

READ: s ← s+1
C(s) ← next item on input; advance read head

PRINT: Print C(s)
s ← s−1

HALT: Stop the machine

7. Array instructions

MAS B,j,n: Make array space
{see section 9.7.1}

AVA: Array variable address
{see sections 9.7.1, 9.7.2}

MMAS B,j,n: Make matrix array space
{see section 9.7.2}

9.10. Bibliographical notes

Much of the material in this chapter is adapted from Randell and Russell (Randell [1964]), who first defined a set of instructions for implementation of Algol 60. Ingerman [1961] gives an algorithm for the rearrangement of an OWN array that is redimensioned in place. Sattley [1961] discusses space allocation and dope vectors for Algol 60 local variables and arrays.

Bauer [1968] describes an Algol 60 implementation in some detail. Berkeley [1964] describes Lisp implementation, of interest for the automatic

allocation of list elements and garbage collection of dead elements. Griffiths [1974a] and Hill [1974] contain a comprehensive review of run-time storage management, with special attention to the management of Algol 68 data structures.

Lee [1967] discusses some special addressing problems that arise in Fortran compilers, in particular, allocation considering equivalences. McKeeman [1970] contains complete source for the XPL compiler, a derivative of Algol 60 designed for compiler and systems programming.

Concurrent garbage collection was first proposed by Steele [1975], and later refined by Dijkstra [1976a] and Gries [1977].

Wirth [1971a] describes a Pascal implementation on a CDC 6000 system. His implementation could be adapted to any computer system, however. Wirth [1976c] also gives a complete compiler for a small language, written in Pascal.

CHAPTER 10

OBJECT CODE AND MACHINE ARCHITECTURES

10.1. Introduction

The object code of a compiler can take a variety of different forms: (1) another high-level language, (2) a macro language intended for a macro assembler, (3) a symbolic assembly language, (4) a special intermediate language (IL), (5) relocatable loader tables, with symbolic code references, (6) absolute loader tables, or (7) absolute machine code.

The object code sections of a compiler are, as a rule, more difficult and complicated to design for the latter cases than for the former.

We will not discuss cases (1), (2), or (3) in this chapter. Such a compiler is essentially a string processor and may often be implemented directly through a SDTS as described in chapter 7. Much depends on the ability of the object language to express the concepts of the source language. It may be necessary to develop special algorithms in the compiler in order to get around any limitations in the object language. For example, if the object language is symbolic assembly language and the source is a block-structured language, name conversions will be needed, since most assemblers support only one block level.

Some compilers generate a special intermediate language (IL) that can be further translated to object code on two or more different machines. The intermediate language is usually a simple translation of the abstract syntax trees generated by the compiler. Certain optimizations can be performed directly on an intermediate language, however, in general IL optimization is difficult.

Most commercial compilers generate linking relocating loader tables. *Linking* refers to a capability of connecting procedure calls with their procedures, or variable references with their external declarations, after compilation. This capability permits a user to compile only a few procedures at a time and collect them all together later. The advantage of linking after compiling is that a large program may be brought together in manageable pieces, and errors may be repaired with a minimum of compilation. Without a linking loader, the entire program must be compiled after even the slightest source change.

Relocation refers to a capability of locating a program at an arbitrary position in physical memory just before execution. Clearly, linking requires relocation, since the procedures are generally assigned contiguous portions of memory after compilation is complete. Relocation is essential in a multiprocessing system, so that several independent programs can be executed in a time-shared manner without necessarily unloading and reloading memory for each program. The distinct programs can be assigned to different regions of memory, and the processor can then be rapidly switched from one program to another.

In this chapter, we shall review some intermediate code structures, review three machine architectures and their loader tables, and then close with a description of a generalized code generator system. Some special optimizations for code generation, register allocation, and code improvement are discussed in chapter 11.

10.2. Intermediate Languages

An intermediate language, or IL, usually consists of a relatively small number of simple operations, combined to form a list semantically equivalent to the source program. The operations are chosen to express the source language needs completely, with as little redundancy as possible, and are organized in a simple uniform manner. An IL is not intended to be written or read by a person, and therefore needs no syntactic window-dressing.

Some common IL's are

- Quadruples (or quads).
- Triples.
- Postfix.
- Prefix.

A *quad* expresses a single unary or binary operation as follows:

<operation>, <operand1>, <operand2>, <result>

where the <operation> is applied to the two operands to yield the <result>. The result and the operands are references to some attribute data structure and may be declared program variables or temporary variables. Some other section of the compiler is responsible for assigning physical memory to the result and operands; indeed, some temporaries may be assigned to registers and not to physical memory. The IL is then a sequential list of quads.

For example, the expression A*B+C*D can be represented as the quad sequence

```
*, A, B, T1       {T1 is a temporary result}
*, C, D, T2       {T2 is another temporary}
+, T1, T2, T3     {expression result is T3}
```

Note that we assign temporary cell names as needed; a separate algorithm can determine the number of temporaries actually needed and assign registers and memory.

A *triple* is essentially a quad, except that its result is referred to by the location of the triple, rather than explicitly. Every triple is assumed to return some result, and its result can be referred to in some later triple. The advantage of triple notation is that explicit temporary names are unnecessary. In triple notation, the expression A*B+C*D can be expressed as

```
(1)  *, A, B
(2)  *, C, D
(3)  +, (1), (2)
```

Here, the (1) in the third triple refers to the result generated by the first triple. Of course, a literal "1" must be distinguishable from a triple number.

Some examples of operations that can be expressed as a list of quads or triples are

- Arithmetic operations.
- Math functions.
- Assignments.
- Indexing.
- Comparisons (result is a truth value).
- Logical operators.
- Branches, conditional or unconditional.
- Conversion of real to integer, or vice versa.
- Begin or end a block.
- Procedure or function call.
- Allocation of array storage.

Declarations and control structures may also be translated to quads or triples. A declaration may require space allocation and the fixing of an address label on the stack. Control structures are usually broken into a sequence of statements separated by branches. A case statement requires a few special quads to express a branching table.

Indexing for array or structure access can be handled by indexed "load" or "assign" quads:

```
XL, A, I, T    {indexed load; equivalent to T:=A[I]}
XS, T, A, I    {indexed store: A[I] := T}
```

Then the statement C:=A[i,B[j]] can be translated to the quads

```
*, i, d1, T1     {d1 is the first dimension for A}
XL, B, j, T2     {T2 := B[j]}
+, T1, T2, T3 {T3 is final index for A}
XL, A, T3, T4 {T4 = A[i,B[j]] }
:=, T4, , C      {assignment is a unary operation}
```

Quads or triples can easily be generated from a binary tree. The tree is scanned in pre-order, and the internal nodes are numbered in the usual left-to-right evaluation order. Each internal node N then translates to a triple whose number is the node number, whose operation is the operator associated with node N, and whose operands are node N's children. Thus the tree of figure 10.1, for the expression A*B+C−D/E translates to the triple list

```
(1)  *, A, B
(2)  +, (1), C
(3)  /, D, E
(4)  −, (2), (3)
```

A source program can clearly be expressed as a list of quads or triples. Some reductions on a list are possible without a knowledge of the target machine architecture. For example, constant arithmetic may be subsumed by noticing that certain quads perform constant arithmetic; their result may be computed at compile time and substituted into those quads that require the result. We shall discuss such optimizations in chapter 11.

The flow of control through a program can also be easily analyzed through quads or triples; we need only scan the list, looking for branches. The control flow can be traced without regard to the data values of the variables, provided that the conditional branches are to a finite number of known, specified locations, and also provided that no quad can be modified by a run-time operation.

Some machine-dependent optimizations are also possible on quads, e.g., the so-called *peep-hole optimizations*. A *peep-hole optimization* examines a short sequence of contiguous quads in an attempt to recognize certain patterns that can be combined into a single quad with a special operation. For example:

1. {+, I, 1, T1}, {:=, T1, , I} stems from I:=I+1 and might be coded INCM I.

2. A STOR I from some register R, followed by a LOAD I, is fairly common; the LOAD I can be replaced by no operation or by a register-to-register copy.
3. An indexed reference with a constant index can sometimes be replaced by a simple direct reference with an appropriate address.

Postfix and Prefix

The use of postfix or prefix notation is convenient as a linear representation of a tree structure. These are generated by a left-to-right natural order scan of an abstract syntax tree. The leaf operands are written as they are encountered. A prefix expression is obtained by emitting an operator (associated with an internal node) upon first encountering it; a postfix expression is obtained by emitting an operator upon last encountering it. Thus the tree of figure 10.1 can be expressed as

$$- + * A B C / D E$$

in prefix and as

$$A B * C + D E / -$$

in postfix.

No parentheses are required, unless an operator takes an indefinite number of parameters; even then, the first operand of such an operator could be the number of parameters expected.

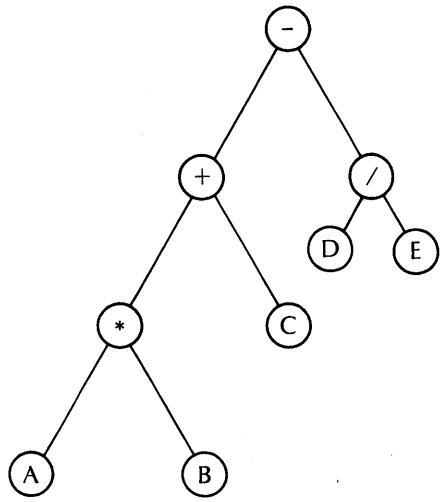


Figure 10.1. An expression tree.

A postfix or prefix string is not convenient for optimization purposes; it must first be transformed into a quad or triple list or into a tree structure.

Exercises

1. Write the following Fortran statements as quads. Use the dimension statements if necessary. Note that in Fortran, the least index corresponding to some dimension is 1.

```
DIMENSION X(16,25),Y(55);
X(3,7) = (I+J+X(I,J)) * Y(5)
I = I*(J+Y(I+5))
```

2. Design triples to support indexed loads and stores, then write the statements of exercise 1 in triple notation.
3. Design triples to support a procedure call with parameters. One triple may be followed (or preceded) by a list of triples that specify the passed parameter values. What can be done about the return value of a procedure? How can call by value and call by reference be distinguished?
4. Sketch an algorithm that accepts a list of triples and then subsumes constant arithmetic and identity operations (e.g., $\{+,25,17\} = 42$, or $\{*,0,(7)\} = 0$) wherever possible under commutativity but not under associativity or distributivity. It should essentially scan the triple list in execution order, evaluate those triples that represent constants or identities, then substitute the results in later triples that require them, evaluating them if possible, etc.
5. Sketch an algorithm that identifies useless triple operations. A useless triple can be removed from the sequence without affecting the semantic validity of the program.

10.3. The Pros and Cons of Intermediate Languages

An IL may be used in a compiler to facilitate optimization or compiler transportability. The former objective will be discussed at length in the next chapter. We shall see that the most convenient intermediate form is a tree or directed acyclic graph. Let us therefore discuss the latter purpose.

Suppose that it were possible to design an IL that could support m

different languages (e.g., Fortran, Basic, Algol). Further suppose that we wish to implement each of these languages on n different target machines (e.g., IBM 360, HP 2100, DEC 10). Without an IL, mn compilers would be required, one for each language and each machine. Some of the software could be shared among the compilers for a particular machine, but in general each of these would be a separate project. Given an IL, on the other hand, only m+n projects would be required—m projects to implement translators from each of the source languages to IL and n projects to implement code generators from the IL to the various target machines. Each compiler then consists of a "front end" that translates source to IL, coupled to a "back end" that translates IL to object code. The mn compilers are constructed (in principle at least) from the m+n components.

Unfortunately, languages and machines are of such diversity that such an undertaking is likely to fail. From the language standpoint, two apparently similar languages are often quite different in detail and require special IL structures to support them. Let us examine just three languages, Basic, Fortran and Algol, as possible candidates for a common IL:

1. The Basic data types are simple numerics, arrays, and strings. The numeric type is usually represented as a floating-point number and is printed as an integer if its fractional part is zero. The Basic strings carry a string count or end of string indicator. Fortran supports several different data types, but not string. Algol supports reals and integers; some Algol implementations also support characters and character arrays. Clearly, an IL must contain memory allocation indicators for each of these types, some of which are used in only one language.
2. The Algol block structuring and procedure call mechanisms are not fully used in Fortran or Basic. Three parameter passing mechanisms must be supported: call by value (Algol), call by reference (Basic and Fortran), and call by name (Algol). These must be matrixed with the data types in a rather complicated way—Basic strings are not passed by value or by name, and Basic strings are not passed in the same way as Algol character strings. The Algol system is the most general, but would be inefficient for Fortran or Basic. Hence the back end should know whether a full Algol system is required or not.
3. Consider the FOR statement (Algol). Comparable structures exist in Basic and Fortran, but have different definitions. For example, the Fortran DO statement is always executed once, regardless of the parameter value viz-a-viz the limit, while the Algol FOR is tested before a first execution. The Algol step size may also be negative, while the Fortran step size can only be positive.
4. Fortran provides a number of special features not found in either of the other two languages: e.g., COMMON, statement functions, equiva-

lences, and special I/O condition keywords. The Fortran equivalence imposes special conditions on the memory allocator that are not found in any other language. We see that memory allocation is radically different in the three languages.

5. Algol has no I/O conventions. Fortran and Basic do support I/O conventions, and the two conventions are quite different. Clearly, a common IL must effectively support two different I/O systems.

These examples of language differences should be sufficient to make our central point: an IL that must support several languages must contain the union set of all those language features, which will make the IL considerably more complicated than it would be for any one of the languages alone. Many of the features must be identified as peculiar to one or another language because of semantic differences. Some language features, particularly the declaration, I/O, and block structuring conventions, require such specialized treatment that there might as well be a different IL for each language.

Some problems arise in the design of back ends with multiple machines and one IL, but they are less severe than the multiple language problem. If the IL is consistently designed, each of its operations can somehow be implemented on a target machine of reasonable capability. Since the IL can in principle be expressed as an abstract syntax tree, there is no reason that efficient code could not be generated from a given IL for several different machines.

Nevertheless, certain optimization problems may occur. For example, array indexing may be supported very efficiently by special mechanisms on one computer but not another. If indexing is expressed by a sequence of primitive triples, with dimension multiplication and addition, etc., it would be difficult to recognize as an operation that could be efficiently performed on one machine. On the other hand, if every such "high-level" operation has a special notation in an IL, then each of the compiler back ends must effectively take them into account, and their complexity will increase. The Cobol language in particular requires a large number of special operations that are directly supported in hardware on some machines and not on others.

Now suppose that an IL is designed with, say, three languages in mind. When a fourth language is to be added to the IL at some later date, the existing IL will likely have to be augmented with some new features. Then each of the back ends must be extended to support the new IL features. This incremental effort will likely be less than that needed for a new compiler, but will not be insignificant either.

The design and implementation of an IL system is a cost in addition to those incurred in a more conventional compiler design. High-level design effort is often more costly than coding, since competent software designers demand higher salaries than coders.

On the surface, an IL is best suited to a multipass compiler; however, it can also be used in a one-pass or interactive compiler. In the latter case, the IL is

simply a software data structure through which all the information must flow in passing from source to object code.

We conclude that an IL may be practical for a few closely related languages and several different target machines. When practical, its use promotes transportability and reduced compiler writing costs, at the loss of some compile-time and run-time efficiency.

10.4. Machine architectures

There are several advantages to a compiler that generates target machine code directly, rather than through an IL. Such a compiler will usually be more efficient and generate more efficient object code. Of course, if a compiler is funded by a computer manufacturer, the manufacturer obviously will be interested only in its machines as target machines. A compiler that can generate code for both its own and its competitor's machines is not likely to be greeted with enthusiasm.

For whatever reason, many compilers are expected to generate instructions tailored to a specific target machine, and that machine's architecture is often not under control of the compiler writer. Of course, a back end of an IL is tailored to a specific machine. We therefore direct our attention to some issues in machine architecture and their relation to language translators.

A compiler that generates specific target machine code seldom generates only that; usually, a set of loader tables must be generated. These tables contain sequences of target machine instructions, but also carry information needed to link separately compiled program segments together, or to relocate the machine instructions after compilation. Relocation data is essential for multiprogrammed target machines that do not handle relocation automatically in its hardware. The IBM 360 is an example of a machine that requires attention to relocation in its translators; the CDC 6400 and HP 3000 are examples of machines that do not.

Certain features of a target machine system are of special interest to the designer of the code generators of a compiler:

1. *The loader table structure.* Only by generating loader tables can the resulting programs enjoy the advantages of relocation and partial compilation.
2. *The register and operating system conventions.* The IBM 360 requires the observance of certain conventions of every program to be executed in its system. Failure to observe these conventions will result in bounds violations or inability of the operating system to diagnose program problems.
3. *Data and program addressing.* How are data and program accessed in the system? Sometimes the addressing convention is very simple, e.g., the

CDC 6400, in which every memory reference instruction carries a full 18-bit address field. In other machines, the addressing convention may be rather complicated and may require special attention.As we have seen for the AOC machine, effective addressing for an Algol-like language requires a set of base registers (the display), instructions that carry an offset from a selected base register, and optional indexing on top of the base register contents and offset. Heap management and access requires double indirection through address labels. Uniform address labels whose nature can be inferred at run-time are very desirable for procedure calls. When such features are missing in the machine hardware, their equivalent must be organized through in-line instruction sequences or procedure calls, often with a sizable loss in performance.Fortran requires much less addressing structure than Algol and can be efficiently supported on most machine architectures. Of course, Fortran does not support recursive procedure calls, nested blocks, or dynamically alterable arrays.

4. *Completeness and uniformity of the instruction set.* Suppose that the system supports four data types. Are all four arithmetic operations supported on all four data types? Are they supported in a uniform manner, or are there special exceptions? Every exception to a general rule requires special attention in some part of the compiler. An operation that only applies to a subset of the class of data types will often have to be supported through a run-time procedure or in-line code sequence for the nonsupported types.
5. *Special instructions.* Often, the instruction set contains some instructions that are technically unnecessary for completeness but that can be exploited for the sake of coding efficiency. An increment-memory instruction is an example.

The designer of a code generator must be intimately familiar with the machine's instruction set. If he is not familiar with the machine, he should spend some time writing assembly language programs and studying the instruction coding conventions and the system conventions. Instructions sometimes have undesirable side-effects or may not perform exactly as expected. Such seemingly minor considerations must be carefully examined in the light of their use in the compiler.

We cannot possibly review all the known machine architectures and their relation to compiler design; there are simply too many. Instead, we shall review three machine architectures in some detail as case studies. These are the Hewlett-Packard 3000, the Control Data 6000, and the International Business Machines 360 systems. Of these, the HP-3000 will receive the most attention; it is a stack machine that incorporates many of the AOC machine notions and its architecture and loader supports partial compilation and program relocation.

The Control Data and IBM machines are examples of multiregister machines. Neither machine supports a push-down stack directly in their instruction set definitions as does the HP-3000, but both can support the AOC machine features without much loss in performance.

Each of the three machines has certain deficiencies that create difficulties for the designer of a code generator. We shall examine both the strengths and weaknesses of each system only from the point of view of overall compiler effectiveness. We do not claim that this is the only or the best basis for the comparison of computer systems. The evaluation of computer system performance must take many other factors into account and weigh them against the initial and maintenance costs of the system.

10.5. The Hewlett-Packard HP-3000

The Hewlett-Packard 3000 series III computer contains several AOC machine features in its instruction set. It is a general-purpose data processing system, specifically for time-shared multiprogramming applications. It is organized around a data stack and a separate instruction memory. Its basic word size is 16 bits, and it is therefore considered a "minicomputer". However, its instruction set contains many powerful arithmetic and stack manipulation instructions, and its operating system provides sophisticated file management and other services.

Since we are principally interested only in those aspects of the HP-3000 that are of direct concern to a compiler designer, we shall limit our discussion to those features. Further information is given in the HP-3000 Computer System Reference Manual.

10.5.1. Data formats

A variety of numeric data types are supported in the instruction set, including complete arithmetic and 16-bit logical operations on 16-bit words, fixed-point and floating-point arithmetic on 32-bit double words, and arithmetic for a 64-bit floating-point number. Special instructions also provide certain data conversions from one number form to another. Packed decimal arithmetic is also supported. Because of the stack architecture, essentially all the operations are performed on the top elements of the stack. For example, a real multiply causes the floating-point number in locations $\{s-1, s\}$ to be multiplied by a floating-point number in $\{s-3,s-2\}$. The two-word result replaces these four words on the stack.

The data types are indistinguishable at run-time, hence it is the compiler's (or user's) responsibility to keep track of the various data types supported by the source language and to emit the necessary conversion instructions.

Some half-word, or "byte" operations are also provided. Bytes may be

packed two per word, and arranged in arrays. We shall not discuss byte and string operations or the special byte addressing modes available.

All the data types (except byte) must be stored on an even-byte boundary (any 16-bit word boundary), owing to the addressing conventions of the machine. The word "must" should be qualified, inasmuch as it is possible to move an odd-byte variable from memory to an even-byte stack top location for an operation, then back again; but such an operation adds considerably to the operation time.

10.5.2. Memory and Register Organization

The machine's memory organization (figure 10.2) is similar to that of the AOC machine. A program consists of one or more code segments; a code segment contains one or more procedures. Only one code segment is active at any one time, although more inactive code segments belonging to the program may also happen to reside in memory. Generally, inactive code segments are kept in a *program file* in disk and are fetched as needed through procedure calls.

A code segment is delimited by two registers, PB and PL (figure 10.2.) The current instruction location is marked by register P. The system enforces the restriction $PB \leq P \leq PL$ and causes a program abort if this restriction is ever violated.

One data segment is normally provided (figure 10.2), and is defined by the following registers.

- DL: Data limit. Defines the least memory location.
- DB: Data base. Defines the "0" memory location. Negative locations are toward DL and positive locations are toward Z. DB corresponds to location 0 in the AOC machine, or D(0) in the AOC display.
- Z: Stack limit. Defines the largest memory location. This limit is automatically adjusted by the operating system as needed.
- S: Logical top-of-stack. Defines the topmost stack word currently in use. Many of the instructions either directly or indirectly affect S. However, an attempted access of data memory greater than S or less than DL results in a program abort.
- SM: Top-of-stack in memory. Of no concern to a user; the system carries the top four stack elements in fast electronic registers that track the memory contents. This register is used in that system. SM is not accessable through the instruction set.
- Q: Stack marker. On a procedure call, the Q-register is set to point to the new stack marker top. Register Q corresponds to D(b)−1 in the AOC machine. There is no display register set.

A 16-bit index register is also provided for array indexing.

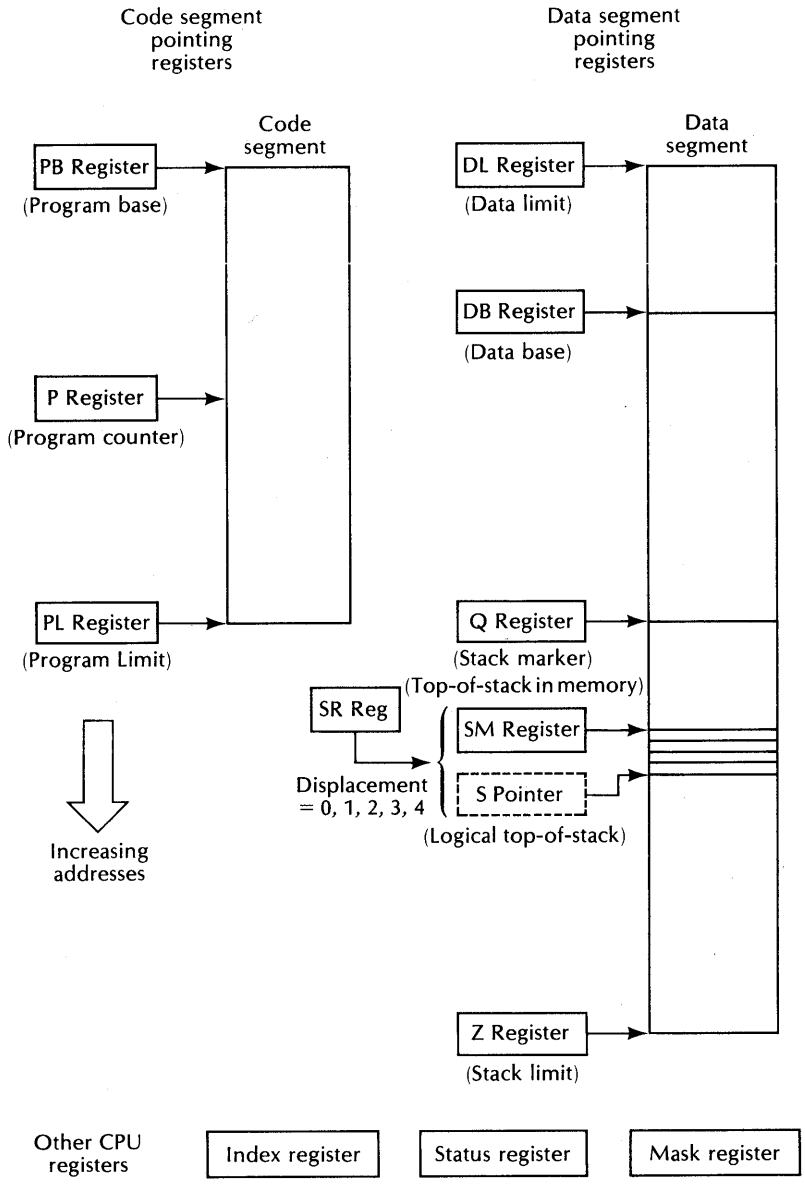


Figure 10.2. HP3000 memory organization and segment-pointing registers.

Address mode	Instruction bits									
	6	7	8	9	10	11	12	13	14	15
P + Relative	0	0	←			Displacement 0:255				→
P - Relative	0	1	←			Displacement 0:255				→
DB + Relative	1	0	←			Displacement 0:255				→
Q + Relative	1	1	0	←		Displacement 0:127				→
Q - Relative	1	1	1	0	←	Displacement 0:63				→
S - Relative	1	1	1	1	←	Displacement 0:63				→

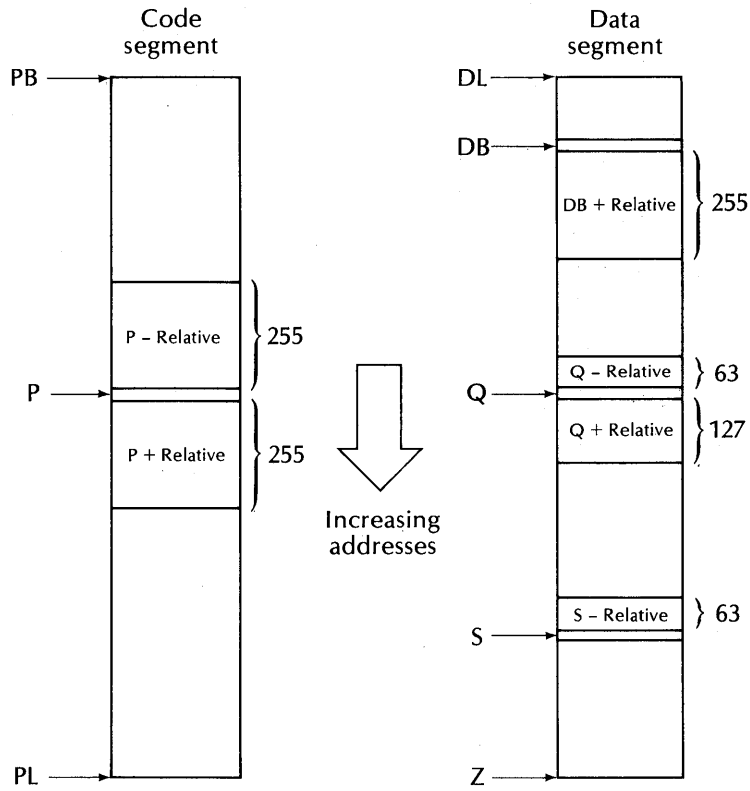


Figure 10.3. HP3000 memory direct addressing modes.

10.5.3. Memory reference instruction format

Twenty-one memory reference instructions are provided in the machine. These have the general form shown in figure 10.3. Bits 6 through 15 of the 16-bit word are used for a direct memory reference, through one of the registers P, DB, Q, or S. Bits 8 through 15 yield a word displacement field for P and DB relative addresses. Bits 9 through 15 yield the displacement for Q+ relative, and bits 10 through 15 yield the displacement for Q− and S−

relative addresses. The directly addressable memory fields are shown in the lower part of figure 10.3.

The DB+ relative field is called *primary DB*. It is used for global or outermost block variables. The primary DB addressing space is only 256 words, hence a large program with many global variables requires an additional capability—indirection or indexing.

The Q− relative field is principally used for formal parameter references. Its maximum size is 63 words, 4 of which are preempted by the stack marker.

The Q+ relative field is used for local variables, at the innermost block level. Its maximum size is 127 words.

The S− relative addressing can be used to access temporary cells whose location is only known relative to the stack top. For example, the RAV instruction in the AOC machine calls for a C(S−1) access.

The P+ and P− relative addressing modes are used in branch and load constant instructions. Access of locations greater than 255 words distant from the instruction requires indirection or indexing.

The maximum code segment size is imposed by the operating system, and also by certain bit fields defined in the system; the upper size limit is 16,384 words. The maximum data segment size (Z−DL) is approximately 32,000 words.

Several code and data segments may reside in physical memory simultaneously. All address references are relative to one of the registers shown in figure 10.3, hence their absolute location in memory is only of concern to the operating system memory manager.

The code segment words are fixed before execution and cannot be changed by a user program. Because of this property, a given code segment can be shared by more than one user in a multiprogramming environment, and a code segment need never be written to disk, once it is completed by the compiler, linker, and loader. (A copy already exists on disk.) A data segment, however, must be written into a scratch area on disk upon transferring control of the system to another user.

10.5.4. Indirection and Indexing

Because the range of direct addressing is so small, indirection and indexing are provided for the memory access (MA) instructions. In general, bit 4 of the instruction indicates indexing, and bit 5 indicates indirection. (There are a few exceptions.) Indirection is to only one level; the direct cell may only point to a data cell, not to another indirect cell. Indirection is illustrated in figure 10.4, in code and data space. For example, LOAD Q+4,I causes a fetch of the word in Q+4 which happens to be 7. The 7 is DB relative, hence the accessed data word is in DB+7. An indirect cell is a 16-bit word, interpreted as a twos complement word count relative to DB. Hence, the negative locations toward DL can be accessed indirectly, as can all the stack area. The

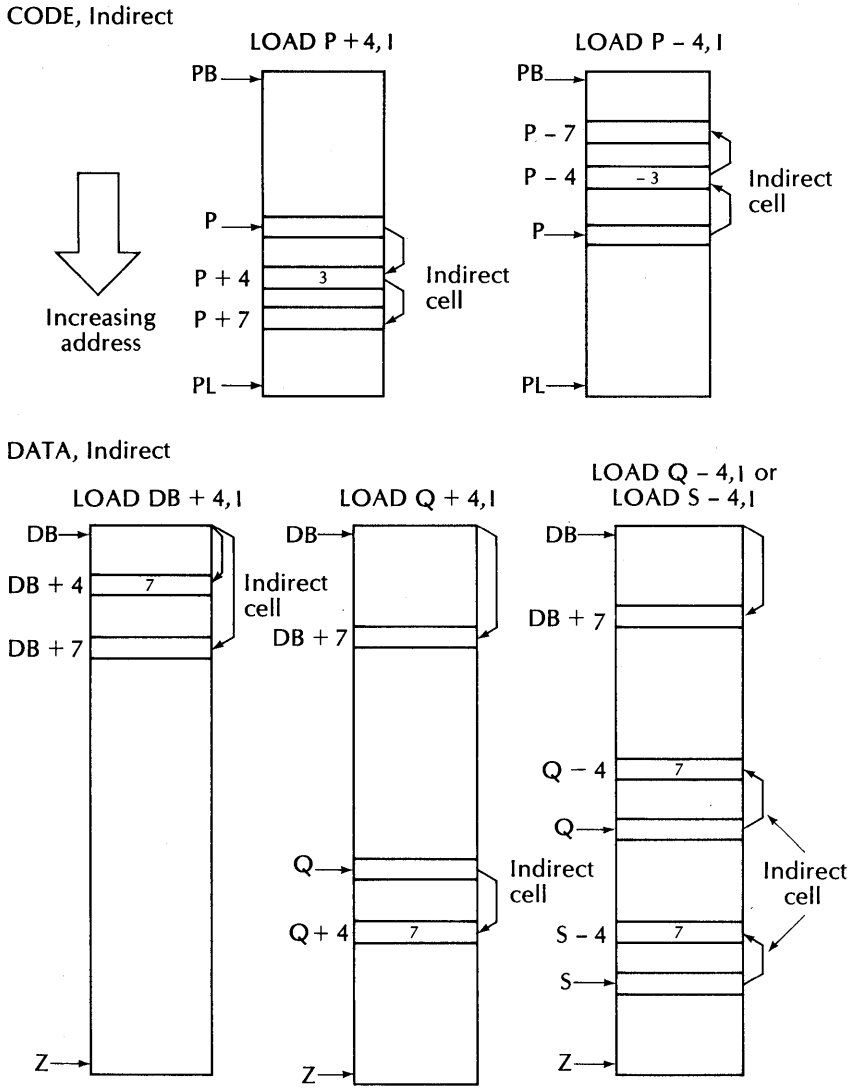


Figure 10.4. Examples of indirect addressing for HP3000.

maximum addressable indirect space is therefore DB ± 32,767. However, the operating system limits the DL to Z size to approximately 32,000 words.

Indexing through the index register X provides an additional level of memory access. Indexing always follows an indirection, if any, and is illustrated in figure 10.5. For example, the instruction LOAD Q+4,I,X first accesses word Q+4, which contains 3; with C(X)=5, the data word at DB+10 (octal) is accessed.

We now see how a data space in excess of 256 words (or whatever) can be allocated and accessed. If a global data area of (say) 500 words is needed for an array, it may be located somewhere above DB (but below the lowest stack marker) and accessed through an indirect word between DB and DB+255. Similarly, local arrays may be allocated above Q somewhere and accessed through an indirect word between Q+1 and Q+127.

Long branches can also be implemented by a short indirect branch through a word in the program memory. An indirection through a word at P+4 that contains "3" refers to location P+7 (see figure 10.4). That is, the indirect cell is relative to itself, just as the direct instruction displacement is relative to itself.

Self-relative instructions and indirections mean that all of the code of a procedure may be fixed by the compiler and need not be changed when the procedure code is positioned arbitrarily within a code segment. (This statement is not quite correct, as we shall see; e.g., procedure calls cannot be fixed by the compiler.)

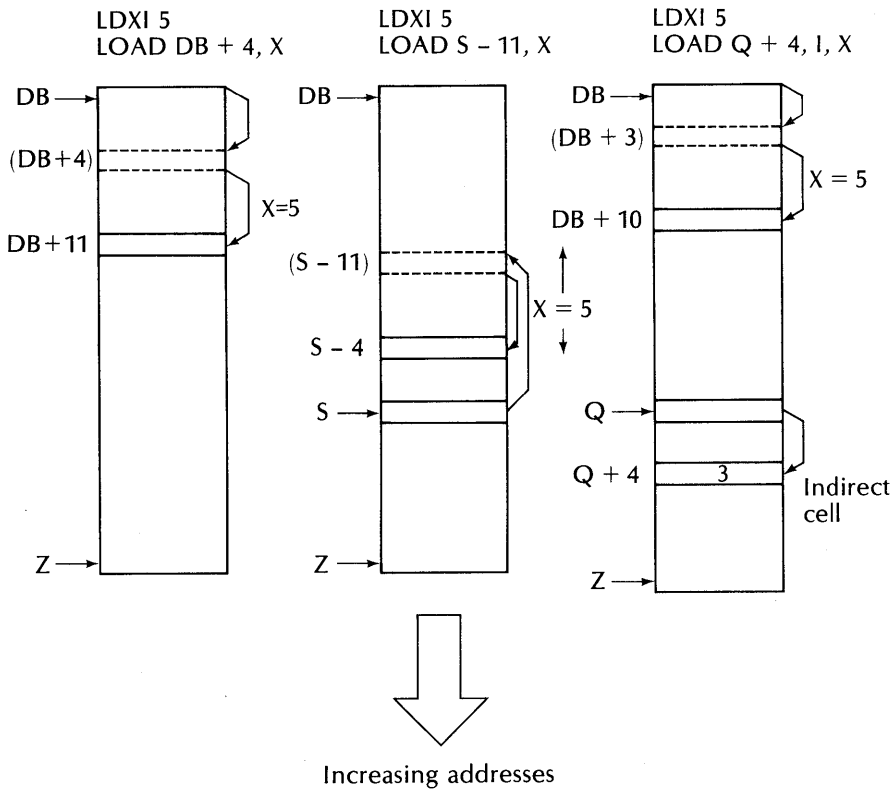


Figure 10.5. Examples of indirect and indexed addressing for HP3000.

10.5.5. Stack Configuration During Program Execution

Figure 10.6 illustrates a typical machine stack configuration during execution. Primary and secondary DB (*global data area*) are organized and initialized prior to execution. An initial stack marker is also placed by the system, and an EXIT through that marker returns the program to control of the operating system.

On a procedure call A, actual parameters are loaded on the stack prior to the call. The call places another stack marker; this marker contains a reference to the previous marker. Q points to the new marker. Local variables may then be built up on the stack. If A calls procedure B, this process is repeated, etc. Three procedures (A, B, and C) are currently active in figure 10.6, and control currently resides in procedure C, called by B. Of course, temporary storage is available between the local variables of one procedure and the actual parameters of the next call.

This machine automatically generates a stack marker, links it to the previous marker, and sets Q, upon executing a PCAL (procedure call) instruction. The PCAL also performs certain other useful services, as we shall see. Upon executing an EXIT instruction, Q is reset to the previous stack marker, and the stack is cut back.

10.5.6. Stack Marker, Procedure Calls and Exits

The stack marker is shown in figure 10.7. It carries the index register (X) contents at the moment of call, a PB-relative return address, some status information, a code segment number (more about that next), and a reference to the previous marker ("Delta Q").

By writing C(X) in the stack marker and subsequently restoring it upon a return, C(X) is effectively unchanged by a procedure call.

Among the status information is "O", an arithmetic overflow bit, "C", an arithmetic carry bit, and "CC", a condition code. The condition code CC may carry one of three values 0, 1, 2 (3 is not used). Code CC is set by a variety of instructions, and is tested in one form of conditional branch.

We now show how the HP-3000 system handles procedure calls and exits. Recall that all the code for some procedure resides in a code segment, that a program may consist of several code segments, and that several procedures may be placed in one segment.

The system permits procedures to be moved around from one code segment to another without recompiling the procedures. This is an important performance consideration, as we shall see, and is well worth the complicated mechanism that makes it possible.

Every procedure call is made through a *segment transfer table*, or STT (figure 10.8). One STT resides at the end of each code segment. It is a directory to the procedures. There is one STT entry, a *procedure label*, for

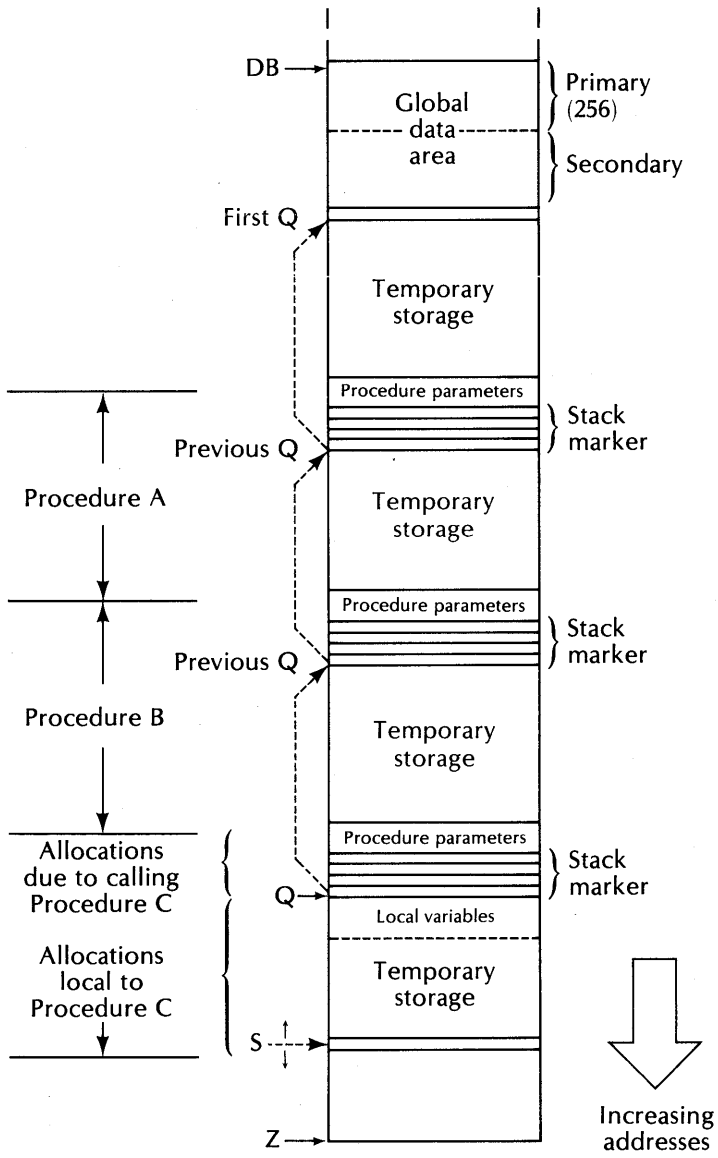


Figure 10.6. Typical stack configuration.

each procedure called in the segment. Different calls to the same procedure share the same STT entry. The STT entries are indexed backwards in memory, starting at the last word of their host code segment.

The PCAL instruction carries an index N ($1 < N < 255$) into the STT table. STT(0) carries the largest STT number, not a procedure label.

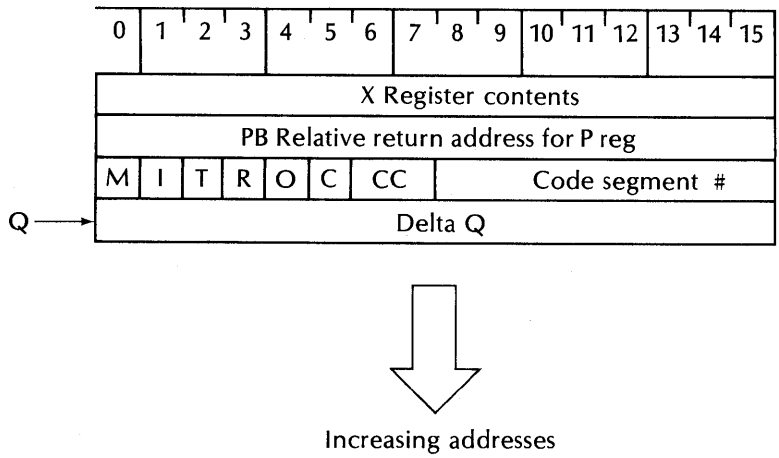


Figure 10.7. Stack marker format.

A procedure label has two forms, depending on whether its procedure is in its own code segment or not. Bit 0 specifies the form (see figure 10.9 for the general format.) Bit 1 (*uncallable* bit) is used for protection of operating systems procedures against nonprivileged calls, thus a 14-bit PB-relative address is available for a procedure call. If the procedure being called is in this code segment (*local program*), then the label carries its PB-relative entry location; a transfer to that location, along with the usual stack marker creation, etc., is made. In figure 10.8, the PCAL(4) in code segment 23 (path 5) is to a local procedure, whose label is in STT index 4. Thus control passes to some code location in segment 23 (see path 6).

If the procedure is in some other code segment (*external* procedure), there are two possibilities, shown by the PCAL(5) in code segment 23, to a procedure in code segment 22. Segment 22 may or may not currently reside in memory. (It may have been overwritten by the memory manager in order to provide space for other code or data memory segments.) This information is in a master table called the *code segment table* (CST). The CST is shared by all users of the system and is strictly managed by the operating system. It is accessed through a CST pointer in location 0 of absolute memory.

The CST entries are double words; their structure is shown in figure 10.9. Bit A is set if the code segment is absent. When it is absent, the memory manager must be called into service to allocate enough contiguous space for the segment (the length field is part of the CST entry). When space is available, an absolute disk address in the CST entry is accessed to load the new code segment. The old segment is marked "inactive," hence its space becomes available. When the new segment has been fully written to memory, the system transfers control to the procedure entry point, and adjusts the

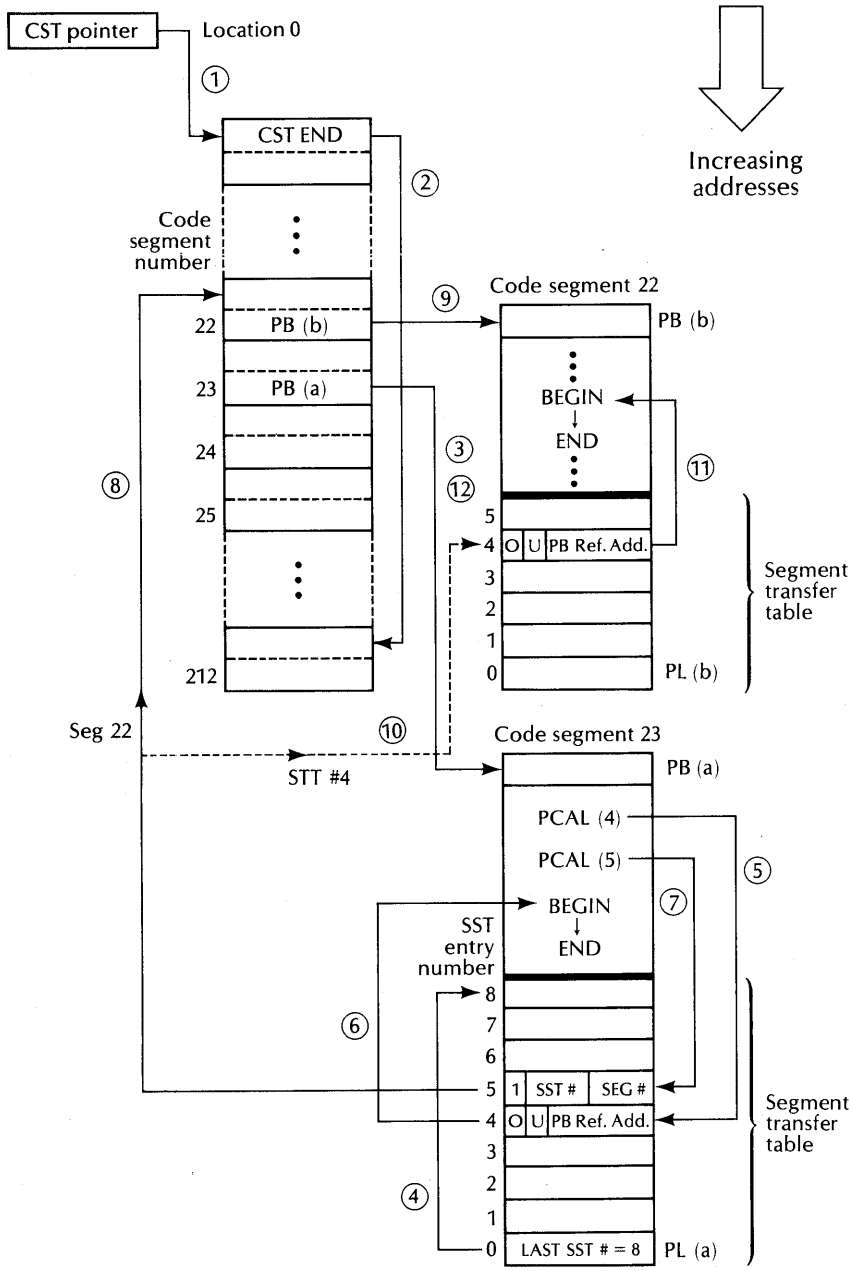
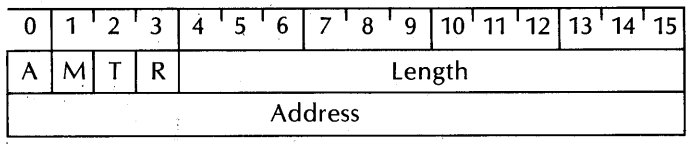


Figure 10.8. Procedure call and exit system.

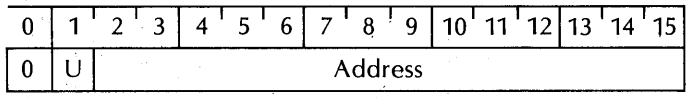
Code segment table Doubleword



A Absence bit (= 1 if segment is absent)
M Mode bit (= 1 if priveleged mode)
T Trace bit (= 1 to call Trace routine)
R Reference bit (for statistical use by operating system, set to 1 when accessed)
Length This value times 4 (max = 16,380)
Address Absolute memory address (for PB) or absolute disc address if absent

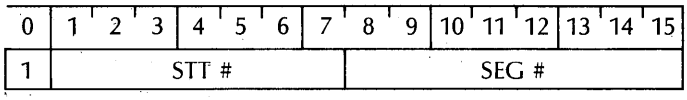
Segment Transfer Table Words

Local Program Label



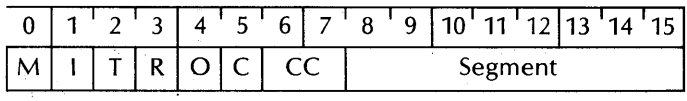
U Uncallable bit
Address PB relative, + only

External program label



STT # STT entry number in target segment, maximum = 127
SEG # Target segment

Status Word



M Mode bit (= 1 for privileged mode)
I Interrupt enable (1)/disable (0), external
T Traps enable (1)/disable (0), user
R Right Stack Opcode bit (pending = 1)
O Overflow bit
C Carry bit
CC Condition Code
Segment # currently executing

Figure 10.9. Formats associated with code segments.

CST entry appropriately. The ADDRESS field of the CST entry is changed to become the new code segment's memory address.

Since several procedures may reside in the same code segment, several CST entries may be affected by the disk fetch operation.

In figure 10.9, the path followed by a call to an absent procedure starts with the PCAL(5), then through paths 7, 8, 9, 10 and 11.

If the procedure is present in memory (through being left around, or someone else's use of it), the CST entry so indicates this fact, and the time-consuming process of fetching the segment from disk is obviated.

Note that an external procedure STT label (i.e. a procedure not in this code segment) carries an STT number in the target code segment as well as the target segment number. Also note that the stack marker for the currently executing procedure contains the caller's segment number, and the absolute PB return location. Upon an exit, the caller's code segment might still be in memory, or might have been overwritten, and therefore must be replaced from disk. Once in memory, the return transfer is easily made by using the PB-relative return address and the CST number held in the stack marker (figure 10.7.)

Of course, upon switching code segments, the PB and PL registers associated with the program must be changed and P set appropriately.

This entire algorithm is embedded in the PCAL and EXIT instructions of the machine. The programmer need not be concerned with any of this complicated mechanism. We discuss it, because it has an important bearing on the structure of the files that a language compiler for this system should generate, the *unsegmented library* (USL) file.

10.5.7. Instructions

There are 13 instruction formats in this machine's architecture, six of which are illustrated in figure 10.10. We shall consider only a few of the instructions, omitting those that are of interest only to an operating systems programmer, those that are of marginal value in a compiler, and those that are essentially repetitions of those presented. We thereby hope to provide some flavor of the architecture of this machine.

Sixteen bits is a marginal instruction size for a machine of this sophistication. As we have seen, the MA (memory address) instructions have a severely limited direct address range, and there cannot be very many of them, owing to the small op-code field. Nevertheless, a large number of operations and data types can be supported through the stack architecture. Stack operations require no address field, only an operation; there can be many of them without a correspondingly large instruction field for them.

The HP-3000 stack-ops are 6 bits each and may be packed two per word. There are 64 of them, a few of which are defined next.

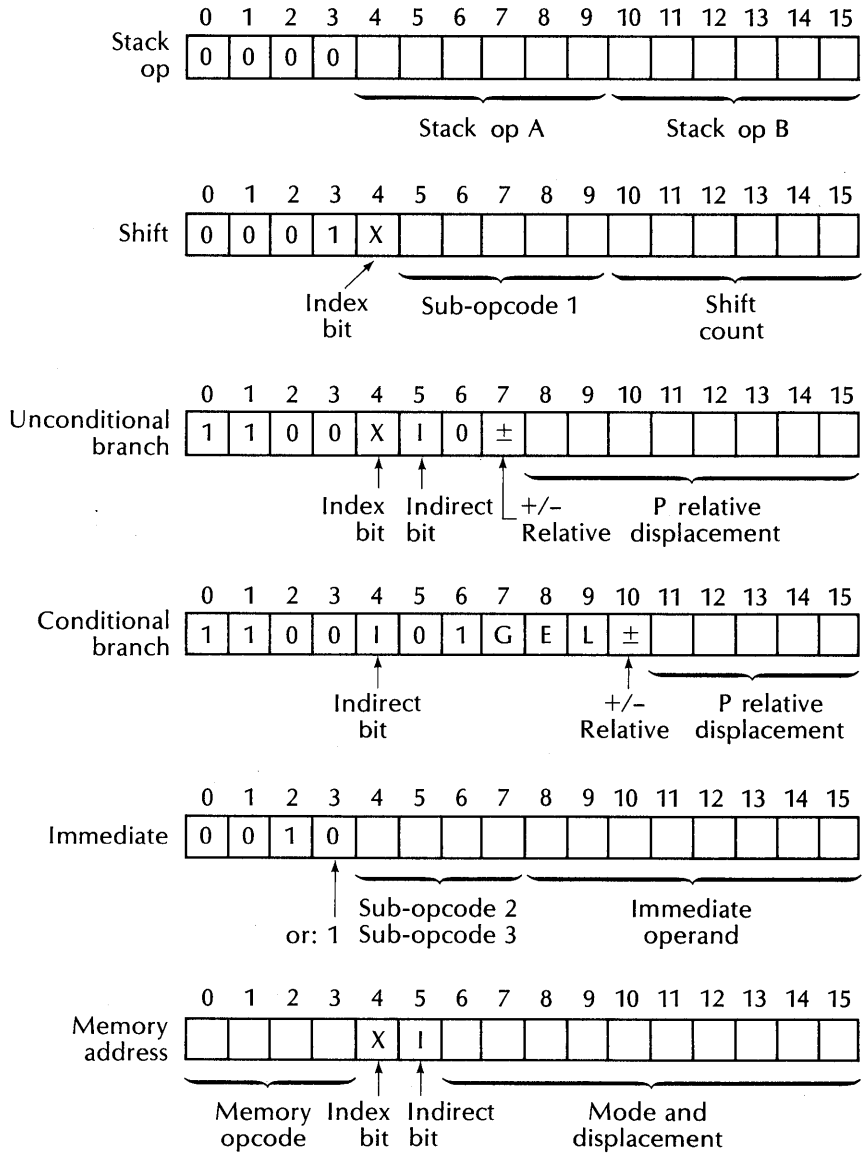


Figure 10.10. Instruction formats.

- ADD, SUB, MPY, DIV are binary 16-bit 2's complement integer operations with overflow. They of course operate on the two top-of-stack words, and replace them by one top-of-stack result word.
- LADD, LSUB, LMPY, LDIV are binary 16-bit 2's complement integer operations with rollover, but no overflow.

- DADD, DSUB are binary 32-bit 2's complement integer operations. DMPY and DDIV are available, but are not stackops.
- FADD, FSUB, FMPY, FDIV are binary 32-bit floating-point operations.
- OR, XOR, AND are binary bit-by-bit logical operations on the top two stack words.
- NOT complements the bits of the top stack word.
- NEG, DNEG, FNEG negates the TOS element—an integer, a double integer, or a floating-point value.
- FLT converts the top integer into an arithmetically equivalent floating-point 32-bit number.
- FIXR, FIXT converts the top 32-bit floating-point number to a 16-bit integer. FIXR rounds the number, and FIXT truncates the fractional part.
- CMP, LCMP, DCMP, FCMP compares the two TOS values (2 integers, 2 logical integers, 2 double integers, or 2 reals) arithmetically, deletes them, and sets the condition code CC correspondingly. CC=0 means A>B, CC=1 means A<B, and CC=2 means A=B, where B is in (TOS) and A is in (TOS-1).
- ZERO pushes a 1-word zero.
- DZRO pushes a 2-word zero.
- DEL deletes the top-of-stack (TOS) word.
- DDEL deletes a pair of TOS words.
- DELB deletes the word just under the TOS word.
- XCH exchanges the top two stack words.
- DUP duplicates the top word on the stack.
- DDUP duplicates the top word pair on the stack.
- CAB rotates the top three words on the stack. If they are initially A, B, C, with (TOS) = A, then CAB causes the order C, A, B.
- NOP, no operation, is used to pad a stackop to form a full instruction word.
- INCX, DECX increments (decrements) the index register.
- ZROX clears the index register to 0.
- STBX sets the index register to the word in (TOS − 1), without affecting the stack.
- STAX sets the index register to the word in TOS and deletes TOS.
- LDXA pushes the current value in the index register.

A branch to some instruction is always to a full word, hence any statement

to which a branch may occur cannot begin on an odd-position stackop. This is a reasonable restriction, but one that can be bothersome to design into a compiler.

The *immediate* instructions are listed below. Each of these carries a eight-bit field containing an operand n, where $0 \leq n \leq 255$.

- LDI n, push n on the stack.
- LDXI n, replace (X) by n.
- CMPI n, compare top TOS word to n, set condition code, delete TOS.
- ADDI n, add n to top TOS value.
- SUBI n, subtract n from top TOS value.
- MPYI n, multiply top TOS value by n.
- DIVI n, divide top TOS value by n.
- LDNI n, push $-n$ on the stack.
- LDXN n, replace (X) by $-n$.
- CMPN n, compare TOS word to $-n$, set condition code, delete TOS.
- ADDS n, allocate n words on the stack.
- SUBS n, remove n words from the stack. *Note:* if $n=0$ in the ADDS or SUBS instruction, the count is taken to be (TOS).
- ADXI n, add n to (X), replacing (X).
- SBXI n, subtract n from (X), replacing (X).
- ORI n, form logical bit-by-bit OR of n and (TOS), replacing TOS. Leading 8 bits are taken to be zero.
- ANDI n, form logical bit-by-bit AND of n and (TOS), replacing TOS.
- XORI n, form logical bit-by-bit exclusive-or of n and (TOS), replacing TOS.

These are expensive in instruction space, inasmuch as 8 bits are needed in each one for the immediate operand. However, the bulk of the literals found in common programs are small integers, and the immediate instructions obviously improve the machine's performance when they can be used. Only integer immediates are provided. All double-integer and floating-point numbers must be stored somewhere and fetched through a MA instruction. However, instructions and execution time can sometimes be reduced by using an integer immediate instruction and converting the result to the desired type.

The memory address (MA) instructions are given below. The symbol "P" means that the instruction supports P-relative addressing as well as DB, Q and S relative addressing.

- LOAD (P) fetches a word and pushes it.
- TBA, MTBA, TBX, MTBX (P) are only P-relative instructions, used to control a FOR-loop with integer index and an integer step size.

- STOR pops the TOS word to memory.
- CMPM (P) compares the TOS word to a memory word, setting the condition code and deleting TOS.
- ADDM (P) adds the memory word to TOS, result to TOS.
- SUBM (P) subtracts the memory word from TOS, result to TOS.
- MPYM (P) multiplies the memory word by TOS, result to TOS.
- INCM, DECM increments or decrements the memory word by 1. TOS is unaffected.
- LDX (P) loads X with the memory word. TOS is unaffected.
- LDB loads a byte in memory to TOS. If indirect or indexed, this expects a byte label and/or byte index offset.
- LDD (P) loads a double word to TOS.
- STB stores a byte in TOS to memory. If indirect or indexed, this expects a byte label and/or byte index offset.
- STD stores a double word on TOS to memory.
- LRA (P) generates a one-word address and pushes it. The address points to the datum specified by the address field, indirection and indexing. If P-relative, LRA generates a PB-relative word address.

Although a stack machine really only needs LOAD, STOR, and branch MA instructions, several more are provided in the HP-3000 for the sake of program efficiency. These are expensive instructions in terms of instruction bit space; the mode and displacement use 10 bits, and indexing and indirection use another 2 bits. This leaves only 4 bits for an operation code, and of these sixteen combinations, four are needed for the non-MA instructions. However, some doubling-up is possible. The STOR instructions cannot apply to P-relative addresses, so that bit 6 can be used to distinguish two store instructions. Using this trick and some others, the system designers have fit 21 distinct MA instructions in this group.

The LRA instruction is similar to the LA instruction in the AOC machine. It may accept indirection and indexing and yields a DB or PB relative address. A PB address on the stack is of limited value in this system; it is needed only for certain operating systems functions and for certain high-level vector operations that we shall not discuss.

The INCM and DECM MA instructions are valuable as optimizations. A typical program may contain several statements of the form

```
I:=I+1
```

or

```
I:=I-1
```

and these are obviously translatable to INCM I and DECM I, respectively.

The LOAD and STOR instructions have counterparts in each of the other data types, except LONG floating-point (four words). There is a LDB and STB (for bytes), a LDD and STD (for double words). Unfortunately, the long floating-point data type is not supported very effectively. Long arithmetic is performed through three addresses on the stack—result, operand 1, operand 2. This arithmetic philosophy is fundamentally different from that for the other data types and requires an elaborate compiler system to support.

The unconditional branch instruction BR (figure 10.10) can be indirect and indexed, but since indirection precedes indexing, a BR L1,I,X is essentially useless. The branch reach is limited to P$\pm$ 255. A larger branch reach must be indirect, with the indirect word within the 255 word limit.

The conditional branch BCC (figure 10.10) is not indexable but may be indirect. The bits GEL refer to the condition code possibilities greater-than (G), equal (E), or less-than (L). The bits GEL may be set in any pattern to select one of six possible conditional branch cases (the remaining two cases yield a "never-branch" and an "always-branch" condition.) The reach of the BCC is only P$\pm$ 31. It turns out that a fairly large number of branches are within this range of words. Those outside this range must be accessed through an indirect word.

The condition code (CC) used in the BCC instruction are set by a variety of arithmetic and stack top operations. CC is affected by some, but not all, the instructions that influence the stack top. Note that the compare operations have no other effect than setting CC. The use of a single CC register rather than the stack top is a weakness in this architecture, but one that can be surmounted if necessary; the CC and other status information can be pushed if it must be saved and later used. Recall that in the AOC machine, the result of a comparison was a TRUE or FALSE pushed on the stack.

The shift instructions (figure 10.10) accept a 6-bit shift count, and the shift count is added to (X), if the X bit is set. Single and double word shifts are provided.

The PCAL and EXIT instructions have been discussed previously. Each carries an 8-bit field, with value N ($0 \leq N \leq 255$). Recall that N in the PCAL refers to a word count from the end of the code segment backwards, and that if N is zero, a procedure label is expected on the top of the stack. (It is for this reason that index 0 in the STT is not used for a procedure label.) The EXIT carries a count N ranging from 0 to 255; N is the number of words below the stack marker that are to be deleted upon the return. Thus EXIT is used to remove the formal parameters from the stack. It is the compiler's responsibility to determine N in the EXIT. However, a compiler cannot determine N in the PCAL, nor organize an STT.

The instruction LLBL causes a procedure label (copy of the STT entry) to be loaded on the stack. Like the PCAL, it refers to the STT bottom-up through a displacement N.

10.5.8. Allocation of Memory Space to OWN and Outer Block Variables

OWN and outer block variables in an Algol-like language belong in some data area that is unaffected by the procedure calls and exits. In the HP-3000, a reasonable static OWN area is secondary DB, just below the initial stack marker (figure 10.11). A dynamic OWN area and space for dynamically allocated arrays (like PL/I VARYING arrays) could be allocated in DL to DB; however, we shall see that there are some problems with this in this architecture.

Now the assignment of static OWN space can be made by the compiler only if every one of the procedures utilizing OWN is compiled. A fundamental premise of the HP-3000 system is that partial compilations should be provided for every language for which it is feasible. This means that any given compilation may be only of some of the procedures with OWN; consequently, the compiler will not know how to assign OWN space. This task is left to a program called a *segmenter* that does have complete procedure information.

The system also provides for additions to primary DB and secondary DB from different procedures. (However, this capability is not invoked in all the supported languages.) Hence, in general, the initial stack configuration of figure 10.11 is not determined by the compiler but through special tables intended for the segmenter.

10.5.9. Partial Compilation and Segmentation

Partial compilation implies another capability that must be present in the system—it should be possible for a procedure that has been previously

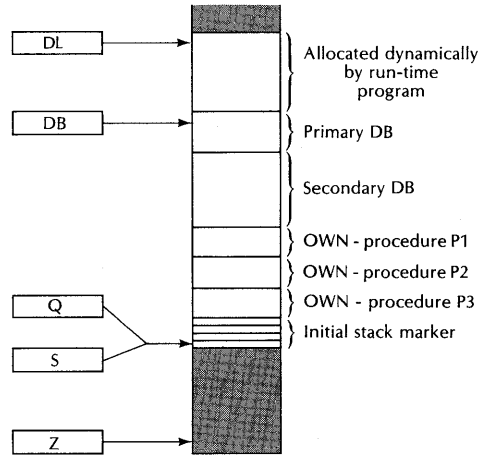


Figure 10.11. Stack configuration just before program execution.

compiled to be overwritten with a new procedure without affecting the others already written. This capability should work whether the new procedure contains more or less code, etc., than the new one. This is a useful capability, inasmuch as it significantly reduces the compilation time required to modify one or two procedures in a set of several hundred in a large program.

Another very desirable capability is arbitrary movement of procedures from one code segment to another, a process called *resegmentation*. We have seen that an external procedure call can be very expensive—a disk access may be needed, some memory manager work may be required, and lots of bookkeeping is needed. Obviously, procedures that call each other very frequently should be in the same code segment. It is desirable that considerable freedom be provided, after compilation, to move them in the interest of performance. This capability, in fact, is one of the major factors in improving execution performance on this system.

To support these capabilities, an HP-3000 compiler generates a USL (unsegmented library) file. The USL file essentially isolates procedures that are tentatively assigned to code segments but that may be reassigned. The procedures and their calls have not yet been connected. The USL file is processed by the segmenter to yield a program file. The program file is essentially a list of the code segments and the initial data segment for the program. Space for the STT entries is provided but not yet filled in; these tables can only be filled in just prior to execution. Finally, the program file is processed by the *linker* to yield an execution file. The linker is invoked just before execution. It assigns CST numbers, fixes up the STT's, and then launches the program into execution.

10.5.10. The USL File Structure

The USL file consists of a set of procedure *entries* linked to *headers*, to each other, and (through a chain) to segment entries. Figure 10.12 illustrates such a file, containing three segment names, SEG1, SEG2 and SEG3, and four procedures, PROC1, PROC2, PROC3, and PROC4. (This is a simplified version of the actual HP-3000 USL file structure.)

There are four linkage systems, indicated by the directed lines in figure 10.12. The segment links start from a directory node through each of the segment entries and end in a "0" pointer (the pointers are labeled S.L.) Each segment entry must be active, and there cannot be more than one with a given name. The segment names may be defined by the compiler, or they may be defined by the segmenter.

The hash links are through each of the entries (segment or procedure), designated H.L. in the figure. These provide rapid access to any given name in the file; the segmenter must connect procedure calls (in headers) to their procedures (entries) by their names. A hash table scheme is used to locate

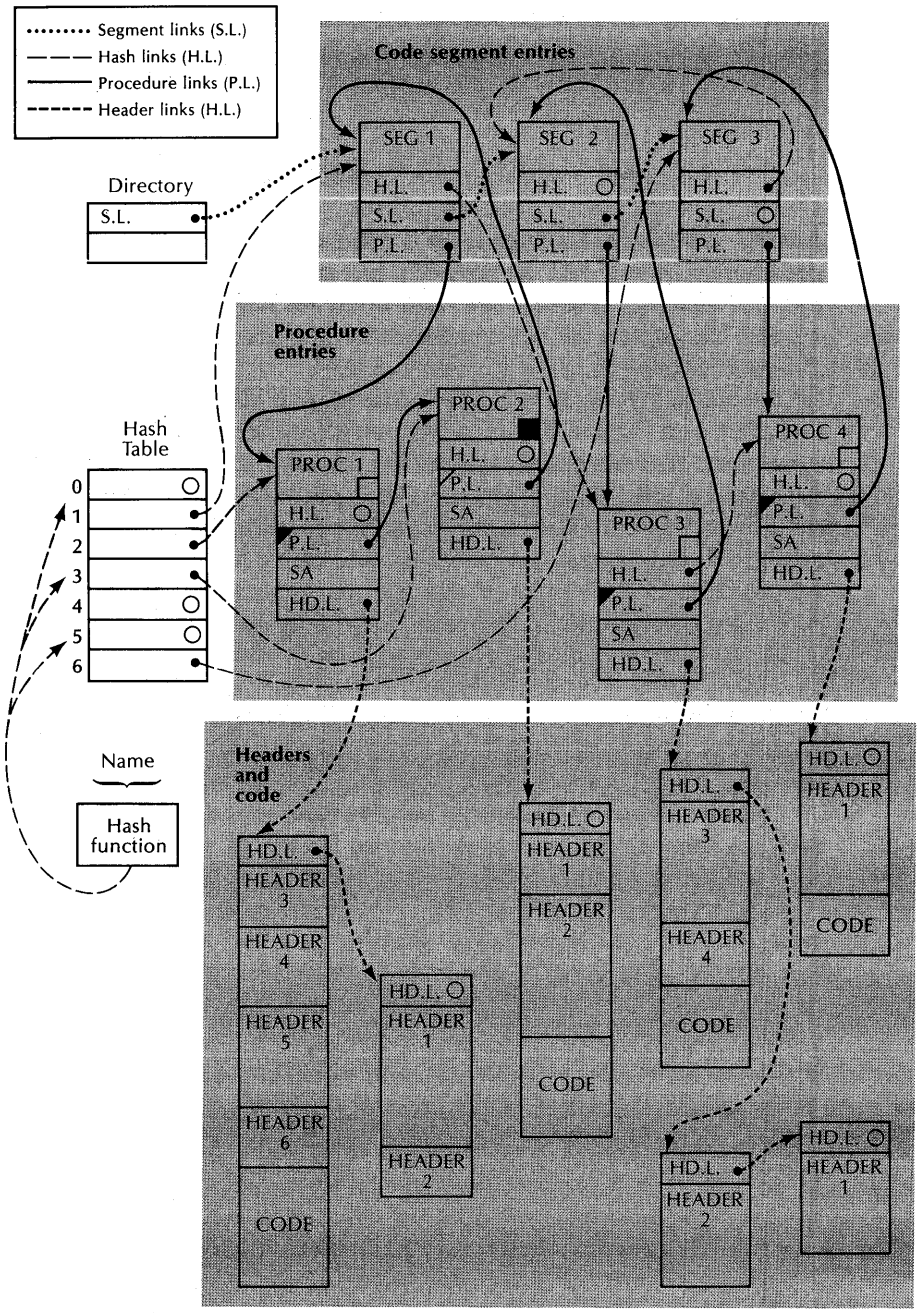


Figure 10.12. Unsegmented library (USL) file structure (simplified).

entries and segments by name; a given name is translated through a hash function into an index in the hash table. The hash chain from the table then links every name in the file with that hash code.

The third set of links are *procedure links*, designated P.L. These form a circular chain from a segment entry through every procedure entry intended to be part of that code segment. A procedure entry may be marked *inactive* (indicated by the open corner in the P.L. field), in which case it has effectively been replaced by another active one (shaded corner) somewhere. More than one procedure with the same name may exist in the USL file, but only one may be active. (The system actually permits exceptions to this rule, for the sake of having the same procedure code in several different code segments, but a discussion would take us too far afield.)

Finally, each procedure entry is linked to a set of *headers* and a *code* block. Some typical headers will be described later. They provide for certain services that the compiler cannot directly provide such as initializing outer block data space, specifying procedure calls, and fixing OWN data space. The code block is a list of the instructions in the procedure.

A header block may consist of several headers. Several header blocks may also be chained together.

It should be obvious that a procedure can be moved from one code segment to another one by merely changing its procedure links. Everything else associated with it is moved with it.

A segment entry is very simple. It only contains a name and several pointers, nothing that would require a new compilation. Hence new ones may be easily created by the segmenter.

The various entries and headers are held in one binary file, and are written as though the file consisted of a single list of words.

10.5.11. Procedure Compilation and USL Linkage

The compiler has a number of tasks to perform upon compiling one procedure, in addition to generating a code sequence for it, as follows (see figure 10.12).

1. Code and headers must be written to the USL file and their location noted. In general, new code and headers are simply added to an existing file by appending them.
2. At the end of a procedure, the hash chain for that procedure is searched for any existing procedures with that name. Any existing ones are marked inactive, but no attempt need be made to reclaim their space.
3. A new entry for the procedure is written to the file and connected to the hash chain at its head. Note that this means that the most recently

entered ones are the first noted by the segmenter in its search for procedure names. The hash table is then modified and updated.

4. The new procedure entry is linked into its procedure chain, again at the head of the chain. For this, the compiler must know the segment in which the new procedure belongs.
5. The headers belonging to the procedure are linked to the procedure entry.
6. The directory contains certain information regarding the available and unused space in the USL file; this information must be updated by rewriting the directory.

The compiler needs the following tables in memory:

- SEG: An image of the current segment entry.
- DIR: An image of the current directory.
- HASH: An image of the current hash table.
- PROC: An image of the current procedure entry.
- HEADER BUFFER: A one-record buffer for headers and code.

The header buffer is essentially filled as headers or code are generated (only one or the other at a time); when full, it is emitted to the USL file as a complete record. At the end of a procedure, it must be flushed to disk. The DIR and PROC tables are of course updated to reflect the written header and code locations and lengths.

At a procedure end, PROC is written to disk and linked into its chains as previously described. SEG is then updated on disk, by either replacing an existing one or adding a new one into the seg and hash links. DIR and HASH are then written to disk; these overwrite the previous ones.

By only updating the USL file in this manner at a procedure end, the USL file is protected against some program error or other serious malfunction in the compiler. There shouldn't be any, but sometimes a user will abort the compiler without thinking about its effect on the USL file. If the compiler is aborted, or a procedure error is detected, no harm is done. Although header information has been written, the file directory on disk is not aware of it, and all the file links reflect a configuration in which the failing procedure has not even been seen.

10.5.12. Primary DB Header

We now turn to the structure of certain of the USL headers. Their form is rather arbitrary, but reflects a solution to the segmentation requirements stated previously.

Every header has a common first word structure containing NW, the

number of words in the header, and a header type code. From these, the length and the type of each header are known. (Note from figure 10.12 that headers can either be arranged in a contiguous list or linked. We have not shown certain details such as how the net length of a list is determined.)

The primary DB header, figure 10.13, specifies the arrangement of the first 255 words (or less) of DB. These may be absolute values or pointers into secondary DB. Since the segmenter, not the compiler, fixes the location of secondary DB, the values of the pointers cannot be set by the compiler; the compiler can only set the relative offset into the secondary DB area.

The segmenter must be told which of the primary DB cells are absolute values and which are pointers; that is the purpose of the array U_i, $0 \leq i \leq N-1$, where N primary words are specified. The value N can be inferred from the number of words in the block NW.

If $U_i = 0$, then C(i) is unchanged; this would be an absolute value. If $U_i = 2$, then C(i) must be incremented by SDBA by the segmenter; SDBA is the DB-relative origin of secondary DB.

The size of this header block is variable, but cannot be greater than 289 words because of the 256-word limit of primary DB.

10.5.13. Secondary DB/OWN Initial Values Header

Initial values for secondary DB or OWN can be specified by the header in figure 10.14. Of course, some languages do not provide initialization as part

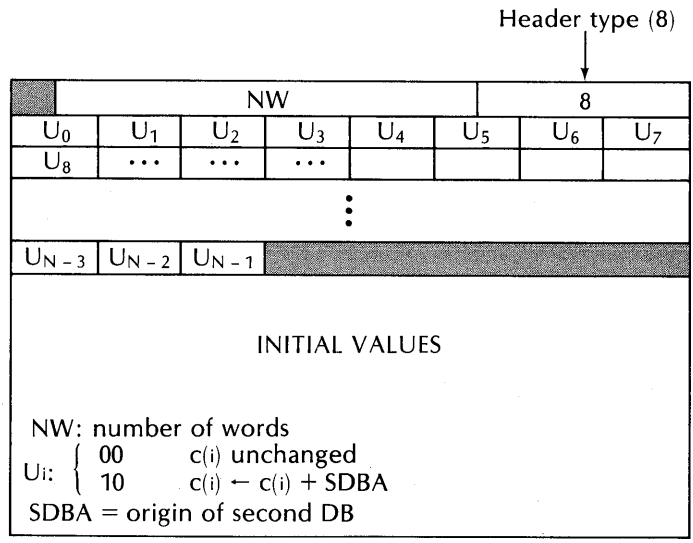


Figure 10.13. Primary DB header.

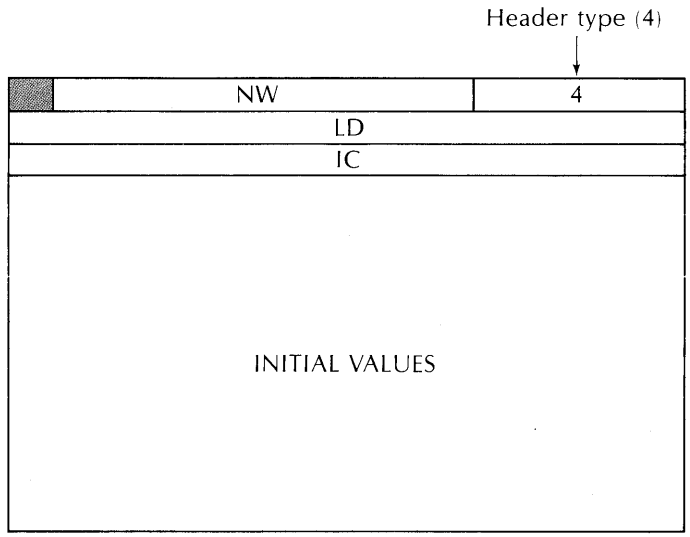


Figure 10.14. Secondary DB and OWN initial values header.

of a declaration; in these, this header would be unnecessary. Also, in principle, the compiler could generate special code executed prior to user code, to initialize the desired data areas. However, the initialization information must be stored somewhere, and it might as well be stored in the USL file as in code.

Parameter LD is a word displacement of the desired initialization data in the secondary DB or OWN area, regarding the first of the area as having displacement 0. The segmenter must then write the initialization words in DB+LD+SDBA, for DB, or DB+LD+OWNBA, for an OWN. SDBA is the DB-relative address of secondary DB. OWNBA is the base address of an OWN area assigned to this procedure.

IC is a repetition factor, and it is useful for initializing large arrays with a repetitive sequence of values. For example, a large array might be initialized to zeroes by setting IC to the array size and placing one 0 in the "initial values" section.

10.5.14. Procedure Call Header

Figure 10.15 illustrates a procedure call header and its application. As we have seen, a procedure is called through a PCAL instruction which expects a STT index number as its operand. However, the STT is organized by the segmenter, not the compiler. The STT's will of course change as procedures are moved around among the segments after compilation, so that the compiler has no idea where the procedures will be located, or the STT index for the PCAL instruction.

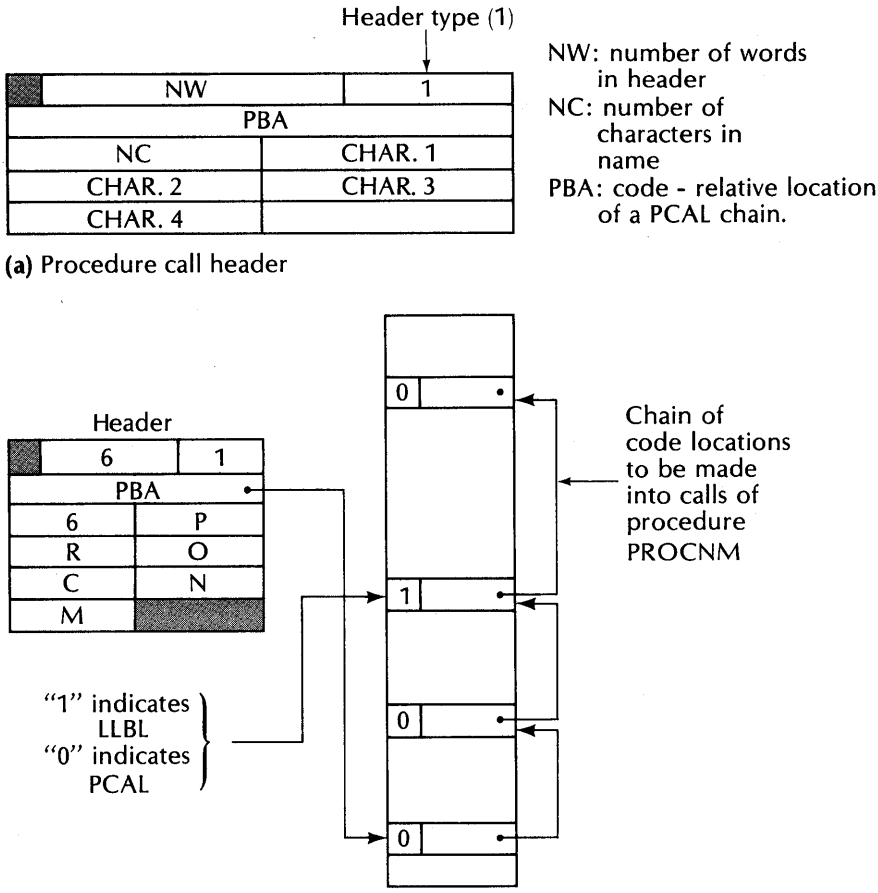


Figure 10.15. Procedure call header and its application.

The PCAL header contains a code-relative index PBA and the procedure name. PBA points to a code location that must receive a PCAL (or LLBL, depending on the sign bit setting). When the compiler writes the PCAL, it places a pointer to another PCAL location. These therefore form a chain pointing backward through the code, linking every PCAL and LLBL associated with one procedure (figure 10.15(b)).

When the file is processed by the segmenter, each word on this chain is replaced by the appropriate PCAL or LLBL instruction to form a completed procedure code list.

Of course, the segmenter locates the procedures through the USL file hash system.

10.5.15. OWN Variable Pointer Correction Header

Figure 10.16 shows a header that deals with another problem related to OWN variables. Given that the location of OWN variables is fixed by the segmenter, something in the run-time code must nevertheless be adjusted to reflect the final location of some OWN variable. Now OWN variables are accessed indirectly through a label that is written on the stack just after a procedure call. The label is placed on the stack by copying some program location; this location must therefore be corrected by the segmenter, once it decides on the location of the OWN data area for that procedure.

PBA is the location in CODE of a word that must be offset by the location of the OWN data area as fixed by the segmenter. See section 10.11 and the discussion of section 10.5.8 for more details.

The code location PBA is therefore corrected by the segmenter as follows:

$$\text{CODE(PBA)} := \text{CODE(PBA)} + \text{OWNBA}$$

where OWNBA is the DB-relative OWN base address for this procedure. Clearly, the compiler must write an offset of the variable in CODE(PBA).

10.5.16. Procedure Local Variables

As figure 10.3 shows, the direct addressing range for Q+n addresses is limited to $n < 128$ words. By using indirection judiciously, the number of words of local variables can be much greater than this.

Here is a simple *local variable allocation rule.*

1. Each simple single-word variable is assigned a direct access cell.
2. Each array variable and each multiword simple variable is assigned one direct access cell to be used as an address, and as many indirect access words (above Q) as needed.

This rule permits a procedure to have as many as 127 distinct local variables, but they may carry more than 127 words total.

Since indirect addresses are relative to DB, not to themselves or Q, and the location of Q will change from one call to the next, the addresses of local arrays will have to be constructed during execution upon each procedure call. The local variables are efficiently set up as follows:

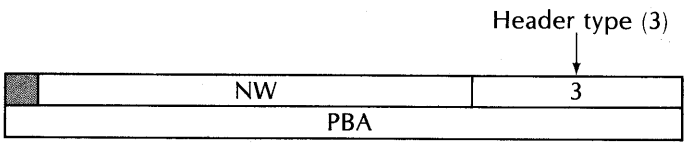


Figure 10.16. OWN variable pointer correction.

1. Upon completion of the PCAL for the procedure, Q and S point to the same stack word, the top stack marker word. If there are any array variables or multiword simple variables, then the instruction

```
LRA  Q+n
```

   is emitted, where n is the Q-relative word address of the first variable. The value of n will be known after all the local variables have been scanned. This writes a DB-relative address label on the stack top.

2. For each array variable or multiword variable containing w words, the instructions

```
DUP
ADDI  w     {or ADDM    =w}
```

   are emitted. These instructions provide another address on the stack, pointing w words past the previous one.

3. If there are m words of simple variables that have not yet been allocated, and p words of arrays and multiword variables, then

```
ADDS  m+p     {or  LOAD  m+p; ADDS 0}
```

   is emitted.

This construction process can be expanded to provide for initialization of some or all of the data spaces. Initialization values are best kept in the code segment of the called procedure and moved to the stack.

Nothing need be done about the local variables upon an exit, as the EXIT instruction effectively deletes all the local variables, the stack marker, and the formal parameters by resetting S.

OWN variables are somewhat more bother. OWN variables must be allocated in secondary DB (see figure 10.11). The most efficient approach is to allocate primary DB space for simple variables or addresses of multiword variables. These DB addresses are only known within the procedure containing the OWN, of course. Unfortunately, primary DB space is limited to 255 words, and this places a severe restriction on the maximum size of programs that can be so supported.

A less efficient approach, but one that conserves primary DB, is to write the OWN addresses on the stack upon entering a procedure. These addresses are not known to the compiler if partial compilation is permitted, hence must be assigned by the segmenter. The scheme makes use of the *OWN variable*

pointer correction header (figure 10.16.) This USL header specifies a code segment location that is to be adjusted to the secondary DB address of some OWN variable. When the program is executed and the address of the OWN variable is needed on the stack, the instruction

LOAD P−m {or LOAD P−m,I}

suffices to load an OWN address. Here, m is the displacement of the code segment address relative to the LOAD instruction. (If the displacement m>255, then an indirect PB label must be used to fetch the OWN word.)

10.5.17. Branches and Constants

Branches have a limited range because of the small number of bits in the partial address field (see figure 10.10). The maximum range is P ± n, where n=255 for an unconditional branch and n=31 for a conditional branch, and where P is the location of the branch instruction. If a branch to an instruction past this range must be coded, then an indirect address word is needed in code; this word must not be executed as an instruction, of course.

Constants are also conveniently placed in a code segment. Three advantages accrue thereby—the constant values cannot be inadvertantly altered through a program error, they are always available (within the same segment as the instruction), and they do not use up valuable DB addresses. Most of the load instructions can be used for PB-relative words, however, the range is limited to ± 255 words relative to the instruction.

A disadvantage in locating constants in code on this machine is that they cannot be passed by reference to a procedure; their address label in a code segment is indistinguishable from a stack address label to the called procedure. The constant must first be loaded on the stack, then its stack address passed. It might as well be given a stack location to begin with in such a case.

We clearly need an efficient algorithm for the allocation of PB space to instructions and *noninstructions* (constants or indirect words). A sketch of a scheme that yields optimal code allocation follows.

We note first that because noninstructions will in general interrupt an instruction sequence and will require a branch around them, we should (1) accumulate a block of noninstructions, to minimize the number of branch-arounds, (2) accumulate no more than 255 words in order to code a direct branch-around, (3) exploit both P+ and P− addressing, and (4) minimize the number of indirect words required.

The process requires two passes. On the first pass, two lists are prepared, a CODE list, consisting only of executable instructions, and a fixup list FX, consisting of blocks of two words each as follows (see figure 10.17)

LC: 1 if instruction refers to an
instruction label.
0 if instruction refers to a constant.
POS: position of instruction in CODE list.
VALUE: the constant value if LC=0, or
the position of the
target instruction if LC=1.

Every P-relative instruction in CODE is assigned an FX entry; the instruction itself with an empty address field is placed in CODE. The FX entries are in the same order as the CODE entries.

At the end of the first pass, we have completed a procedure. Every branch target location is therefore known, and the FX table will therefore be complete.

The second pass is through the FX-CODE list. On this pass, the CODE list will be partitioned into blocks of at least 510 words each. (We assume that all instruction reaches are 255 words, for simplicity.) Between each instruction block will be a dump area DUMP of constants and indirect cells. The general idea is that every P-relative instruction will be able to either reach its target directly or through an indirect cell in a neighboring dump area ahead of or behind the instruction. The target may be another instruction or a dump area word.

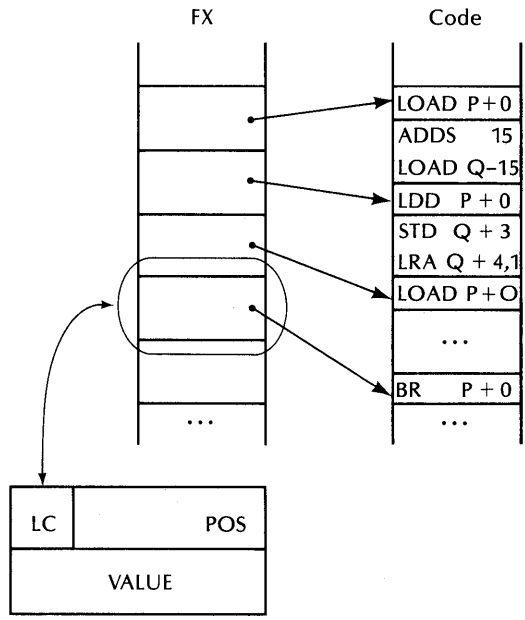


Figure 10.17. Code and fixup list, FX. FX contains an entry for every P-relative instruction in CODE.

The partitioning, assignment of dump words, and fixup of instructions will occur in one general fixup operation. The nature of the operation is suggested by figure 10.18. At a critical point in the fixup, a list of fixed-up instructions and dumps are in a list FCODE, preceding P0. The locations of the instructions up to address P0 are fixed and known, however, a block of instructions INST(i) preceding P0 remains to be fixed up, a dump area DUMP(i) has yet to be determined, and there is some group of instructions BINST(i) whose position is unknown.

The first fixup task, given the situation of figure 10.18, is to determine the size of the DUMP(i) area. We do this by scanning the region of FX that corresponds to INST(i) and BINST(i). Since P0 is fixed and known, the size

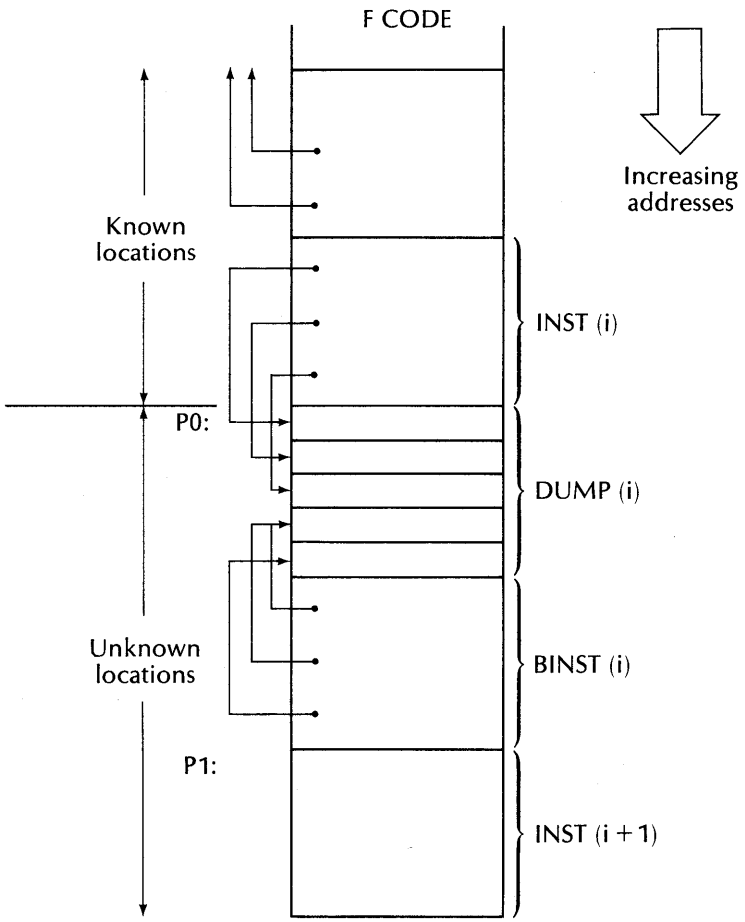


Figure 10.18. Configuration upon beginning to fix DUMP(i) area. We look for constants and instructions that require an indirect word within regions INST(i) and BINST(i); these will be placed in DUMP(i).

of BINST(i) will be 255 words; no instruction located farther than 255 words past DUMP(i) can reach DUMP(i) directly, hence is irrelevant.

Words are placed in DUMP(i) in the order in which they are encountered in the FX table for the FCODE block i. This ordering facilitates the reach situation, since the first instructions in INST(i) will access the first words in DUMP(i), and the last instructions in BINST(i) will access the last words in DUMP(i).

Every constant must be placed in DUMP, however, a multiply referenced constant need only be placed once, provided that each of the references can reach it. Since shared constants are rare, a reasonable plan is to enter every constant as encountered to establish an initial size for DUMP, D.

We next add those indirect words that are essential for branching. To do this, we identify a branch instruction with the maximum reach (given the current DUMP size), which can be achieved through a scan of the FX table for region i. That instruction can either be fixed up with a direct reference or will require a DUMP cell. If it can be fixed up, then all the other instructions can also be fixed up, and no more DUMP words will be needed. If the instruction cannot be directly fixed up, then it needs a DUMP word, and the next largest reach must be found. Instructions that can share an indirect DUMP word with a preceding one can be ignored.

In searching FX for a maximum reach instruction, every location LOC>P0 and position POS>=P0 must be augmented by the current DUMP size D; the required reach R is clearly

R:=ABS((if LOC>P0 then LOC+D else LOC) − (if POS>P0 then POS+D else POS))

If R>255, and an existing word in DUMP cannot be used, then a new DUMP word is need: D:=D+1.

Once the DUMP(i) area is known, the instructions in INST(i) can be fixed up from FX information, the DUMP(i) area can be emitted, and the BINST(i) instructions can be located and fixed up; these will take us as far as location P1. The remaining FX table must also be adjusted to account for the DUMP area size; this is done by adding the dump size D to every VALUE and POS found greater than or equal to P0.

We now must establish the next P0 position. We start with the next FX entry not yet dealt with. If it is a forward or reverse branch with a target within its reach, the instruction is fixed up and we continue the FX table scan. Otherwise, the instruction will require a DUMP word, and we therefore use it as the origin of the INST(i+1) area. The end of the INST(i+1) area, P0, is 255 words farther on, and the position P1 is 511 words farther on. (Some allowance must be made for the branch around the dump area). We are then in a position to repeat the above process.

We see that the overall effect of this algorithm is the clustering of dump

blocks, minimizing their size, and maximizing their separation. The algorithm becomes considerably more complicated if the two different P-relative instruction reaches (31 and 255) are both taken into account. We have also assumed that every constant is a single word. Multiword constants are probably best left to the end of the code and accessed indirectly; however, this is a suboptimal approach in general. Multiword constants could conceivably be placed in dump areas, but the fixup algorithm would become more complex.

10.5.18. Conclusions

It should be apparent that a compiler for this machine must do much more than simply generate a sequence of stack operations for expressions and assignment statements, that is, if partial compilation or resegmentation after compilation is deemed desirable. If neither of these is wanted, then the compiler could generate an absolute loader file and perform its own procedure linkage, STT generation, etc.

In particular, the compiler must generate the USL file structure illustrated in figure 10.12, and be prepared to increment an existing USL file with new information. Certain problems exist with this USL file that introduce compiler difficulties as follows:

1. CODE should be partitioned among several linked blocks rather than appear in one contiguous block in the file. Now it happens that during compilation, code and headers will have to be emitted in an intermixed fashion in general. This means that one or the other must be written to a temporary file or memory area and later copied to the USL file.
2. The initialization header (figure 10.13) has a maximum NW of 1023 words because of its 10 bit field. However, the system can support many more words in a DB or OWN array (up to the limit of 32,000 total for the stack). Therefore, the initialization header generator must be prepared to partition an initialization block for some array into pieces of 1023 words each at most. Furthermore, even a 1023 word header will either require a large memory buffer or must be constructed "on the fly" by sending individual words to a general-purpose header writer.
3. Only one code segment with a given name is permitted on the segment chain, yet several procedures with the same name can appear on the procedure chain. This means that these two entries must be handled in a different fashion. It would be better to treat code segments exactly the same as procedure segments.

These are obviously minor design complaints, and they might have been corrected when the USL structure was designed. However, this structure is

used by several language compilers, and it is impractical to change it. The key point is that the compiler designer must clearly understand the USL rules and provide for its design quirks.

Upon examining the 3000 machine architecture, we find a number of other quibbles that force complications on the compilers as follows:

1. The HP-3000 instruction set is incomplete. For example, memory-reference arithmetic instructions are provided for ADD, SUB and MPY, but not DIV. Also, these instructions are applicable only to integer arithmetic. An optimization to take advantage of them clearly must be accompanied by tests for these conditions.As another example, although long data types are defined and provided for through stack-op instructions (not given here), the instructions expect three addresses on the stack top. This convention is fundamentally different from that of the other data types and obviously requires a special compiler algorithm to support. For one thing, the problem of temporaries crops up. It might be best to simulate the stack-value approach by first allocating stack top space for the two variables, then load their addresses, then execute the instruction.The conversion instructions are also incomplete; however, any conversion can be made with a combination of one, two, or three stackop instructions.
2. The architecture requires three kinds of address labels: a data memory word label, a data memory byte label, and a PB-relative code label. A word label carries a DB-relative word position, a byte label carries a DB-relative byte position, and a PB-relative label carries a PB-relative word position. These are each carried in one 16-bit word and cannot be distinguished at run-time. They can only be distinguished at compile time. Furthermore, a PB-relative label is of no use when passed to a procedure in a different code segment, as no mechanism for accessing the words of the caller's code segment exists, and, of course, the caller's code segment number is lost. The two kinds of data labels, along with the need to organize some arrays through indirect labels, greatly complicates the handling of equivalences.For example, consider an array passed by reference to a procedure. If the actual parameter is a byte array, then the formal parameter should be, too, and a byte label must be passed. Otherwise, a word label should be passed. Clearly, full knowledge of the formal parameter declarations is needed when the procedure call is coded.
3. Only one Q register is available. As we have seen in chapter 9, it would be desirable to have a display for block-structured languages. However, the missing display can easily be compensated for in this architecture. We have two options in dealing with up-level addressing: follow the

static chain to obtain each reference, or follow the chain and write address labels in the local data space. The former is less efficent, while the latter requires two passes of a procedure, one to identify all the up-level calls in the main body of the procedure and a second to generate the code for the necessary address labels.

4. Only one level of indirection is provided. This is a serious handicap in dealing with dynamic arrays. As we have seen in chapter 9, two indirections are needed for the management of a dynamic array space.One might suppose that the following code sequence could be used for a dynamic array address fetch:

```
LOAD  Q+n,I  {load a label}
LDD   S,I,X  {then load the variable—a double word}
```

However, this operation now yields a label on the stack underneath the variable that must be removed. That is not too bad; the instructions

```
CAB, DEL
```

achieve that.The real problem is that it would be desirable to be able to pass dynamic arrays by reference to procedures just as nondynamic arrays are passed. The called procedure needs one uniform way of dealing with them, and this must of necessity be a double indirection for every such array which is clearly inefficient for nondynamic arrays.The only effective solution to this problem is a tag bit on indirect words that effectively says, "I am pointing to another label". This tag is lacking in the 3000, and would be impossible for the manufacturer to add as an architectural extension.

5. Different instructions have different direct address spans. (There are four different effective spans, 255, 127, 63, and 31.) These differences complicate the compiler systems that must keep track of direct addressing limits. The two different P-relative spans (31 and 255) complicate the compiler algorithms that must determine how to place indirect words among the code.
6. The restriction that non-byte variables must be aligned on word boundaries is a problem in such languages as Cobol and Fortran that permit a more general alignment. In Cobol the alignment problems arise through file access; the language permits detailed specification of the location of variables in a file record, and this specification can require alignment of a double-word data variable on an odd byte boundary, for example.In Fortran, alignment problems arise through the EQUIVALENCE statement.

7. The very small number of directly addressable cells available means that special techniques are required to support large programs. For example, many Fortran subprograms contain more than 127 variables. Such programs could not be supported on this architecture if the rule is "one variable per primary word." A way around this limitation is to allocate primary words for variables and pointers, until only one primary word is left. It then is a pointer to an array containing all the remaining data. The compiler must keep track of the offset of the remaining data in this array and use the index register to access data.
8. The code segment size limitation is a restriction on the size of the largest procedure or main program that can be compiled. This restriction can only be lifted by partitioning a large procedure into several smaller ones. Although such a partitioning is conceivably possible automatically through a data flow analysis, in practice only the program writer can do it effectively.

Despite these technical difficulties, it is relatively easy to design a compiler for this architecture. The stack-based arithmetic and procedure control system, and the separation of code and data segments facilitate block-structured languages with recursive procedure calls. Indeed, the systems programming language, designed to access all the machine features efficiently, is a modified version of Algol 60.

Exercises

1. What determines the maximum size of a code segment, other than the stated operating system constraint? What determines the maximum size of a data segment?
2. A procedure may be arbitrarily located within a code segment. Only procedure calls PCAL and load label instructions LLBL need be adjusted by the segmenter in the process. Explain why.
3. Why is a data indirect word DB-relative, rather than Q− or S-relative? (Hint—consider a procedure parameter passed by reference.)
4. Describe how parameters can be passed by value, by reference, and by name in a procedure call. How can a procedure name be passed and later called within the called procedure?
5. Given an Pascal-type procedure declaration, sketch an algorithm that assigns addresses to the formal and local variables.
6. Design an HP-3000 code sequence that writes an up-level address on the top of stack. This code could be used in addition to the local variable stack setup upon entering a procedure.

7. Write a procedure that walks through a binary tree located somewhere in DL to S. The procedure may be recursive. Each tree node consists of two words representing the DB-relative address of the left and the right son node, respectively. A word containing octal 100,000 marks a leaf node. First write the procedure in Algol or Pascal notation, then translate it into HP-3000 assembly language.

8. Indirect words may refer either to bytes or to words. A byte label is the position in bytes relative to the left-most byte in DB0. Given that local declarations may contain both word data types and byte data types, sketch an algorithm that efficiently organizes the stack for the local variables of a procedure. Assume that the local declarations are either simple integers, bytes, fixed integer arrays or fixed byte arrays.

9. Pascal dynamic structures can be organized in DL to DB space on the HP-3000 without the use of double indirection by the following scheme. Each structure is of fixed size, since it is associated with some data type; it is linked under program control to others through pointers saved in the structure. The compiler should then make a list of the different word sizes needed for the dynamic types found in the program, and organize a linked list for each size. Each list will contain some "used" and some "spare" entries in general. A new request can be serviced by allocating one of the spare entries or by allocating a new list element. An entry can be discarded by returning it to the "spare" status. In this way, garbage collection is never needed, and the entries need never be moved, once activated. Sketch an algorithm to support such dynamic structures.

10. Construct a flowchart or design a set of procedures that support procedure calls and exits on the HP-3000. Define the necessary services required of the memory manager and disk file manager.

10.6. The Control Data 6000 Computer System

The Control Data Corporation (CDC) 6000 series is a large, general purpose, data processing system, intended particularly for scientific computing. A major emphasis in its design was placed on fast floating-point arithmetic, almost to the exclusion of string and decimal operations. The Cybernet 7600 systems are also based on the 6000 architecture, but contain a set of string and decimal instructions.

This particular system is of interest for two reasons: (1) the central processor architecture is a good example of a multiregister system with a relatively simple instruction set, and (2) Nicklaus Wirth found that Pascal was fairly easy to implement with the 6000 registers and operations.

The 6000 system consists of three major components—a *central processor* (CP) unit (two in a 6500 system); a large, fast *central memory*; and ten identical *peripheral processor units* (PP's.) Each of the peripheral processors has its own small memory and special instruction set. The central processor has direct access only to central memory. It communicates with the PP's through messages left in central memory, otherwise it and the PP's operate asynchronously during program execution.

The PP's can communicate with the system's data channels and main memory. They can also cause the CP to switch tasks through a special instruction called an *exchange jump*. The PP's are only used for operating systems function, and the CP is essentially only used for user program execution. However, some operating system functions are performed by the CP as special tasks, and all input-output operations must pass through the PP's.

This system is intended for multitasking and multiprogramming. Main memory consists of 60-bit words, with 131,072 words maximum in a system. It may be partitioned into a number of independent contiguous program areas by the operating system memory manager. Not all of the memory is available for a user, however. Some is preempted for the operating system. Data and instructions share the same program space, although a language compiler may of course distinguish the two through software conventions.

Each partition of main memory is controlled through two machine registers, RA and FL (figure 10.19.) RA fixes the base of a program area; every memory reference instruction address is relative to RA. FL fixes the field length. Any attempted access of main memory below RA (possible through an address roll-over) or above RA + FL results in a program abort. FL may be adjusted dynamically through an operating system procedure call.

During time-shared operation of the system, a user program area may be moved to and from mass storage (disk) under control of the operating system. Since there is no distinction between fixed instruction memory and variable data memory, all of the program area is assumed variable, and must be written to disk whenever the memory manager decides to put that job to rest for a while.

An overlay facility is also provided by the operating system to accommodate programs larger than 131,000 words, or to partition a large program into smaller program blocks. However, it is only semiautomatic and requires careful planning on the part of its users.

The central processing unit consists of a set of 24 data registers, a set of one or more instruction registers, and an arithmetic section (figure 10.20). The 6400 and 6500 systems have one instruction register and one arithmetic unit; the 6600 system has an 8-word instruction register queue and ten arithmetic functional units. The 6600 was designed to improve CP performance through parallel processing of arithmetic operations and through providing several instructions in a fast register set. Otherwise, the instruction sets of the three 6000 CPs are identical.

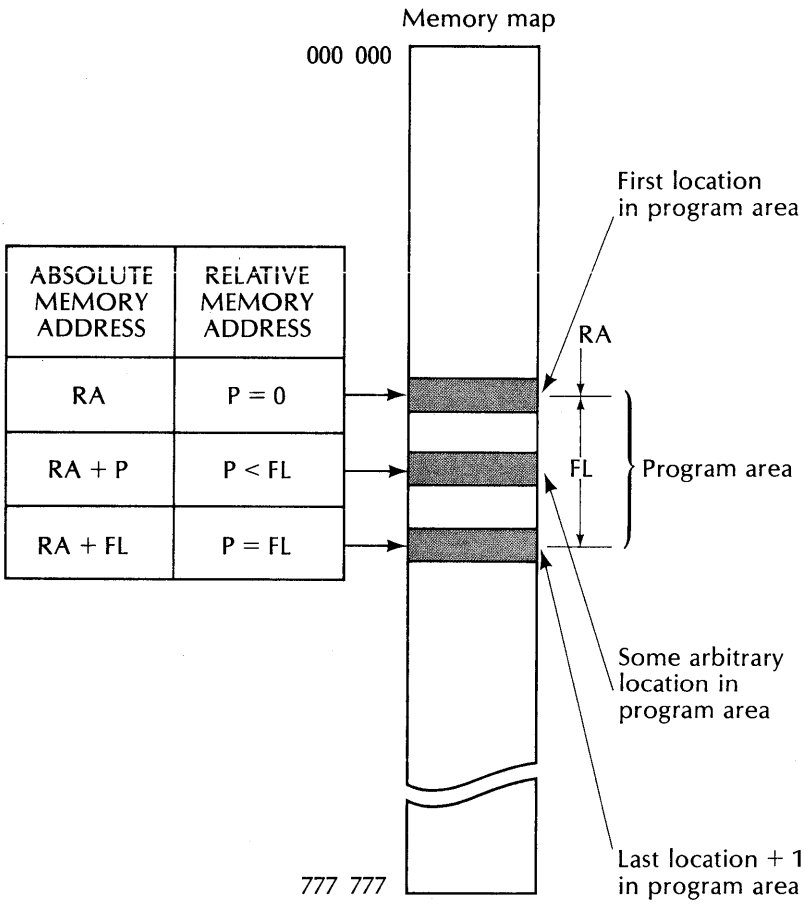


Figure 10.19. CDC6000 main memory organization. Area for one process is RA through RA+FL.

10.6.1. Registers and Arithmetic

An arithmetic or logical operation can only be performed on an operand (or operands) in registers (figure 10.20.) Most of the arithmetic operations use the eight X registers, 60 bits each. The A and B registers are principally used for simple address calculations.

Only full addresses (18 bits) are used in instructions; there are no partial addresses to contend with.

The A registers are coupled to the X registers in an interesting way. Any value deposited in register Ai, where $1 \leq i \leq 5$, causes the contents of memory location Ai to be copied into register Xi. Any address deposited in register Aj, where $6 \leq j \leq 7$, causes the contents of register Xj to be written to memory location Aj. (The new value of Ai or Aj is used as the memory address).

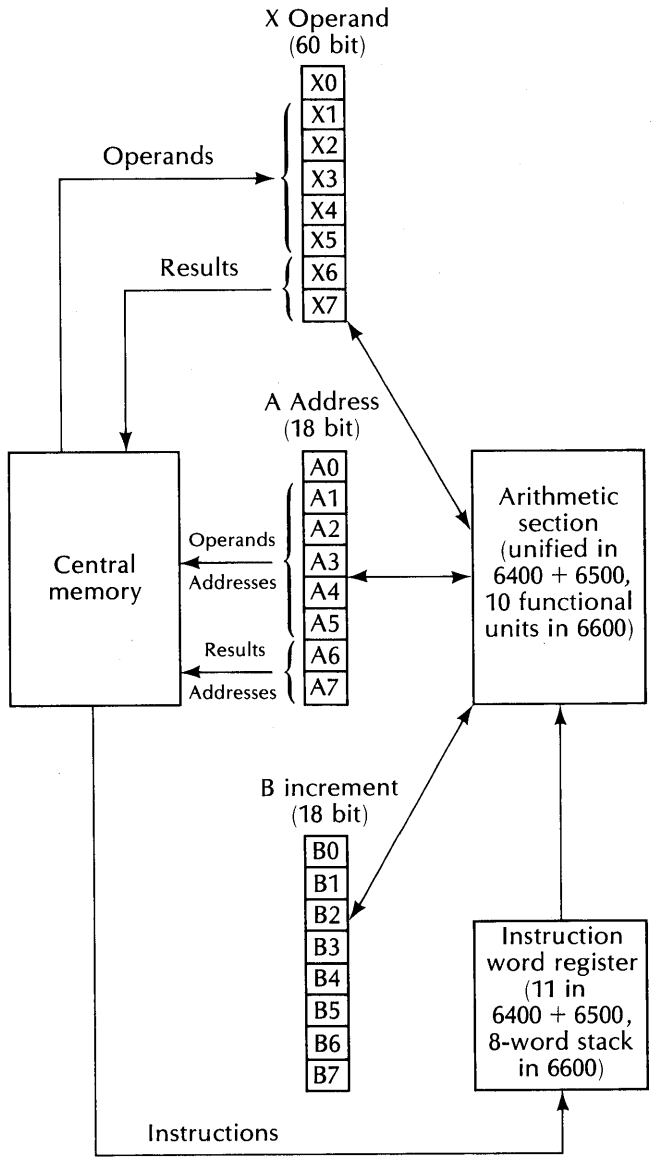


Figure 10.20. CDC6000 register and memory organization.

This X-register loading feature is the principal means the CP has of communicating with main memory.

Register B0 contains zero permanently. Any value stored in B0 disappears, and any reference to B0 is equivalent to a reference to zero.

Registers A0 and X0 are unconnected; no memory access with X0 occurs

when A0 is loaded. A0 can carry any 18-bit value, and X0 any 60-bit value.

The B registers play a prominent role as address offsets in the instructions. They are essentially index registers.

10.6.2. Instruction Format

The CP instructions are either 15 or 30 bits in length. They are normally packed 2, 3, or 4 to a 60-bit word in memory. A NOP instruction (15 bits) is provided for padding, when full packing is impossible. See figure 10.21.

Branches are to the left-most instruction of a 60-bit full word, not to some 15- or 30-bit internal boundary. For this reason, the compilers and assemblers must be sure that a branch target is the left-most instruction in a word.

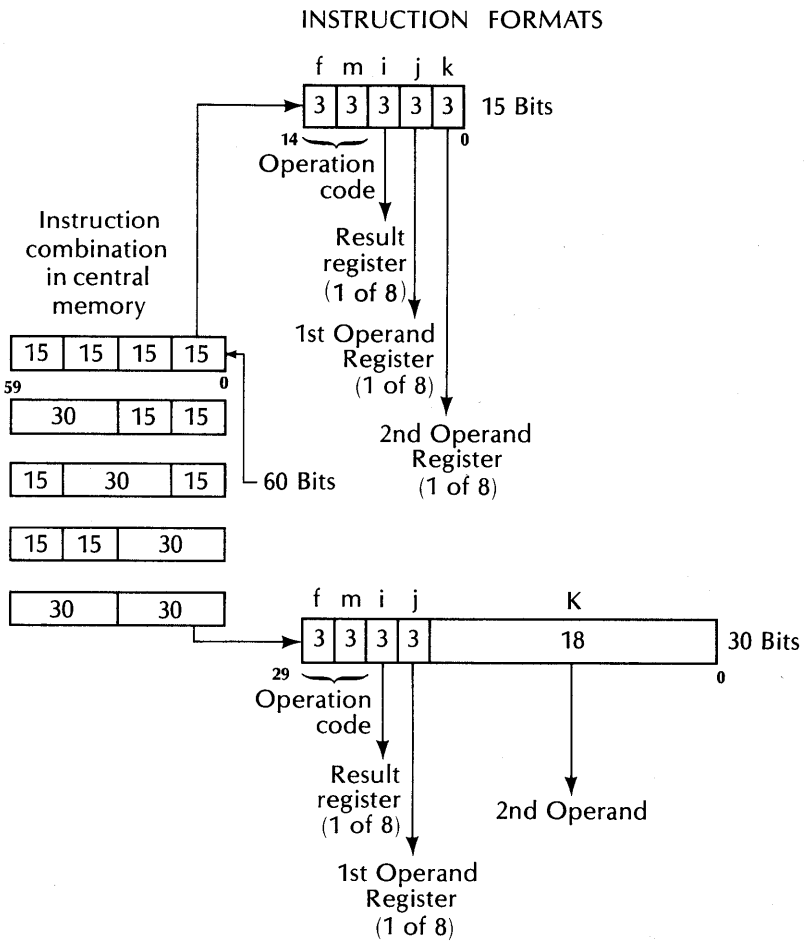


Figure 10.21. CDC6000 instruction formats.

The 15-bit instruction format carries five fields of 3 bits each, f, m, i, j, k. Fields f and m comprise a 6-bit operation code, and fields i, j, and k are register indices in general (there are certain exceptions). The most common 15-bit instruction is a binary operation on two registers j and k, with the result appearing in register i.

The 30-bit instruction format carries an operation code (6 bits), a result register index (3 bits), an operand register index (3 bits), and a full memory operand address K (18 bits).

Every instruction with a "K" reference is 30 bits; all other instructions are 15 bits.

The registers are fully symmetrical, aside from the special A register feature that also accesses memory. Any two registers (whether the same or different registers) can be the operands, and the result can be returned in any of the operand registers. The system copies the source register contents into temporary work registers to perform the necessary operations and then writes the result to the destination register.

10.6.3. The Register Set Operations

The general format of a register *set* operation is below. There are two forms, an "S" form and a "B" form. (COMPASS mnemonics will be used in this discussion, to avoid entanglement with octal instruction codes. COMPASS is the 6000-series assembler.)

$$S \begin{Bmatrix} Ai \\ Bi \\ Xi \end{Bmatrix} \begin{Bmatrix} Aj \\ Bj \\ Xj \end{Bmatrix} \begin{matrix} + \\ - \end{matrix} \begin{Bmatrix} Bk & (15 & \text{bit}) \\ K & (30 & \text{bit}) \end{Bmatrix}$$

```
BXi     Xj
```

The S form can be used to set any of the A, B or X registers with a sum or difference of {A, B, X} and {B, K}. The S form instructions are complete with one exception: Xj − Bk is not provided. Recall that K designates an arbitrary 18-bit address, relative to the program origin RA. Also recall that any SAi (for $1 \leq i \leq 7$) carries a memory access side effect, affecting register Xi or a memory word.

The B form copies one X register to another, e.g.,

```
BX3   X4
```

copies register X4's contents to register X3; X4 is unaffected. When an X register is set with an S-form instruction, only the least significant 18 bits are set; the most significant bits are zeroed.

Some examples of set instructions are:

```
SB1  B3+B4  {sum of C(B3), C(B4)
            replaces C(B1)}
SA2  B2-750  {C(B2)-750 replaces
            replaces C(A2); then C(X2)
            is written to memory address
            C(A2)}
SX0  X0+20   {C(X0) is replaced by C(X0)+20,
            least 18 bits only}
```

10.6.4. Arithmetic and logical operations

Most of the arithmetic and logical operations are performed among the X registers. The general format, in COMPASS form is given below. These are all 15-bit instructions.

$$\left\{\begin{matrix}F\\I\\D\\B\\R\end{matrix}\right\} Xi \qquad [-]Xj \left\{\begin{matrix}+\\-\\*\\/\end{matrix}\right\} Xk$$

Here, F means "floating-point", I means "integer", D means "double precision", R means rounded floating-point, and B means Boolean. The operator + means "add" for the arithmetic operations and logical OR for Boolean. Binary operator "–" means "subtract" for the arithmetics and exclusive-OR for Boolean. Unary operator "–" means "negate" for the arithmetics and logical complement (NOT) for Boolean. Operator "*" means "multiply" for the arithmetics and logical AND for Boolean. Operator "/" means "divide" and is legal only for arithmetic oerations.

The 60-bit numbers can be interpreted in different ways in this system, but the arithmetic units are essentially structured around a floating-point number representation. This representation contains a sign bit, an 11-bit biased exponent and a 48-bit integer coefficient. Double precision arithmetic is performed with two of these, such that one represents the most significant 48 bits of a value and the other the least significant 48 bits.

The real number representation includes special representations of $+\infty$, $-\infty$, and "indefinite." Infinity is obtained whenever a floating-point operation yields a result outside the maximum range of number representation. "Indefinite" is obtained through a mathematically ambiguous operation, e.g., $0/0$ or $\infty - \infty$. Once such special values are obtained, the arithmetic units can continue to operate on them through the usual arithmetic rules. For

example, $\infty + n$ yields ∞, "indefinite"*n yields "indefinite", etc., where n is any finite value. Conditional tests on the infinite and indefinite number forms are also provided.

Rounded ("R") or truncated ("F") single-precision floating-point operations may be selected.

The integer instructions (e.g., IX1 X2+X3) interpret a 60-bit operand as a ones-complement integer. Unfortunately, the machine provides no means of testing overflow on an integer operation, and only integer addition and subtraction are provided. Integer multiplication and division require conversion to floating-point first. Accurate integer division requires several instructions, inasmuch as special attention must be paid to fixing the result correctly.

In ones-complement number representation, the negation of some positive binary number n is formed by complementing every bit. Thus -1 is represented as

11111...110

As a consequence, there are two representations for zero,

00000...000

and

11111...111

Another consequence of this representation is that the least significant bit is 0 for an even positive number and for an odd negative number. Thus an even-odd test based only the least significant bit would fail.

The Boolean operations are the usual bit-by-bit AND, OR, exclusive-or (XOR), and complement (NOT).

Some examples of operation instructions:

```
FX3  X3*X5   {product of C(X3) and C(X5) to C(X3)}
BX4  -X6     {complement of C(X6) to C(X4)}
RX7  X1/X3   {rounded quotient C(X1)/C(X3) to
               C(X7)}
```

10.6.5. Branches

A variety of branch operations are provided as follows. The unconditional branch JP is the only branch whose target can be indexed through a B register. Otherwise, the branch target is address K. The conditional branches ZR, NZ, etc., test the contents of Xj, then branch to location K if the test is satisfied.

JP Bi+K unconditional jump to (Bi+K)

{ZR, NZ, PL, NG, IR, OR, DF, ID} Xi,K conditional jump to (K), based on Xi

{EQ, NE, GE, LT} Bi,Bj,K conditional jump to (K), based on Bi relative to Bj.

RJ K return jump to (K)

Here,

ZR: Register is zero.
NZ: Register is nonzero.
PL: Register is positive or zero.
NG: Register is negative and nonzero.
IR: Register is in range, i.e., not infinity.
OR: Register is out of range, i.e., $+\infty$ or $-\infty$.
DF: Register is defined.
ID: Register is indefinite.
EQ: Two registers are equal.
NE: Two registers are not equal.
GE: Left register $\geq$ right register contents.
LT: Left register $<$ right register.

Note that two branches are needed to test for "negative or zero" or "positive and nonzero."

A pair of B registers may be compared with the EQ, NE, GE, and LT instructions. For example,

```
NE  B3,B5,150
```

causes a branch to 150 if C(B3) $\neq$ C(B5). It turns out that this instruction form can be used as an unconditional branch and is faster than the JP, e.g.,

```
EQ  B0,B0,K
```

unconditionally branches to K.

The return jump, RJ, is useful for Fortran-style, nonrecursive subroutine calls. The mechanism is shown in figure 10.22. Consider the instruction

```
RJ  K
```

in location L. When executed, it writes the instruction

```
JP  L+1
```

in location K, then branches to location K + 1. Location K should contain the first instruction of the subroutine. An exit from the subroutine is achieved by a branch to location K; this branch returns control to location L + 1, just following the RJ instruction.

We use the term "subroutine call" to describe this process, since RJ is totally unsuitable for recursive procedure calls in a block-structured language. A recursive subroutine call using RJ would destroy a previous JP stored in the dedicated memory location. The RJ instruction is not indexable, nor is it capable of storing a location in a register or stack. Its use also renders code nonreenterable; a section of code cannot be used by more than one process at a time.

10.6.6. Other operations

The 6000 system contains several other instruction formats not discussed here. These include shift, normalization, packing, unpacking, ones count, and mask formation instructions. These are of value for number conversion and bit-field logical operations.

10.6.7. Procedure Parameters in a Fortran Implementation

Since recursive subroutine calls are forbidden in Fortran, it is sufficient to allocate registers or memory at compile time for the subroutine local variables and formal parameters. Parameters are always passed by reference, so a list of 18-bit addresses would be sufficient as formal parameters.

The Fortran convention on the 6000 series is to use registers B1 to B6 for the addresses of the first six parameters. The A and X registers are assumed to contain garbage upon entering a subroutine, may be used freely within the subroutine, and need not be restored upon an exit.

If a subroutine contains more than six parameters, the addresses of the extra parameters are written to memory locations just preceding the target of the RJ instruction that calls the subroutine. The arrangement is shown in figure 10.23.

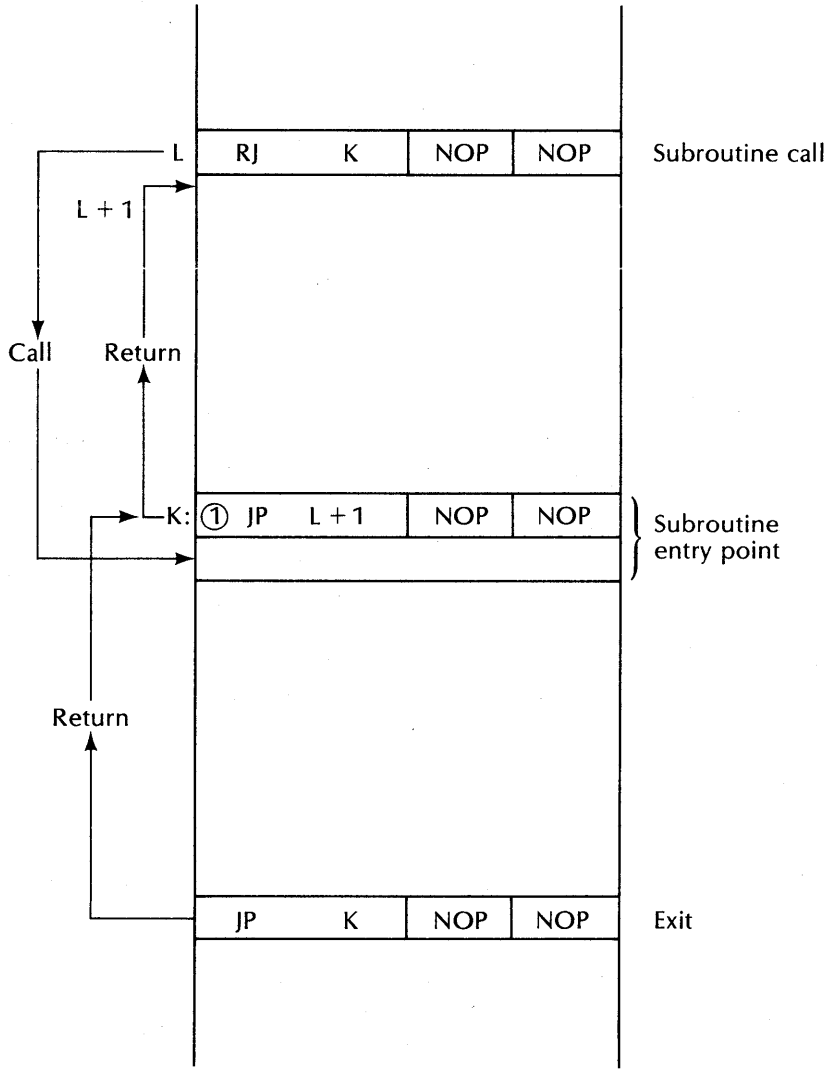


Figure 10.22. Return jump RJ instruction operation.

A typed subroutine returns its value in one of the X registers. No address of the return is necessary.

10.6.8. Arithmetic Expressions

Arithmetic expression evaluation requires a small stack of temporary locations; these could be allocated from the X-register set and memory for the purpose. We can then conceive of a general purpose register allocation system

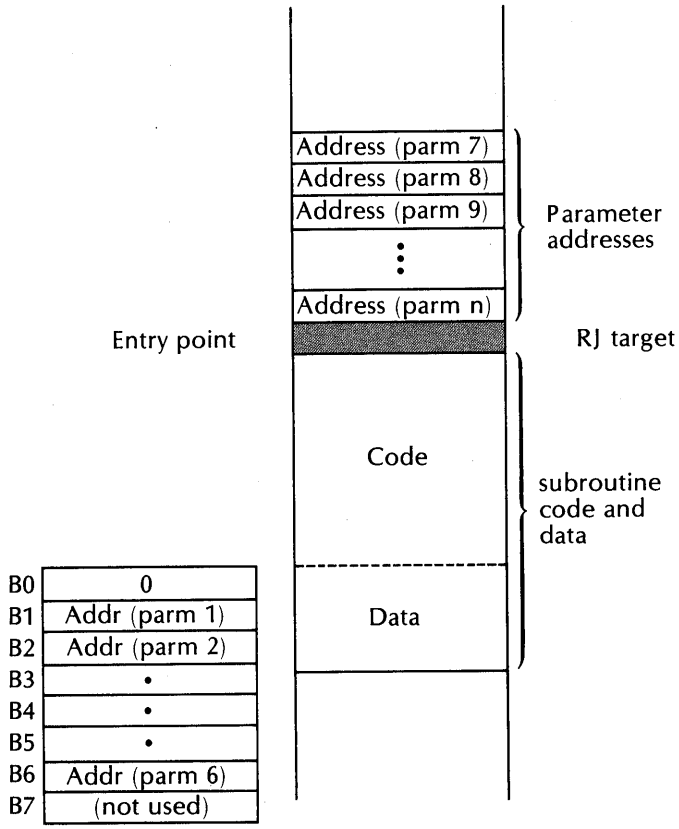


Figure 10.23. Fortran subroutine parameter passing conventions (simplified).

that manages all seven of the X registers. The allocator is of course complicated by the three different roles played by the sets {X0}, {X1 . . . X5}, {X6, X7}. In general, the allocator must attempt to keep variables and temporaries in X-registers as much as possible in the interest of efficiency, yet deal with register allocation by storing some results in order to make room for others. Of course, the instruction form BXi Xj facilitates its housekeeping; an X register can always be stored with two 15-bit instructions provided that X6 or X7 (and A6 or A7) is free.

Chapter 11 discusses some register allocation systems that might be applied to this architecture.

10.6.9. Saving and Restoring Registers

The 6000 system has an interesting deficiency—it is very difficult for a user program to save (and later restore) the contents of all the registers. In saving

the registers, the essential problem is how A6 (or A7) can first be saved without losing vital information. The problem can be solved, but requires a large number of instructions. The key to its solution is a series of 18 operations. Each operation consists of a conditional branch based on the sign of X1 (say), that conditionally causes an RJ and a left shift of X1. The RJ's are used to leave a trail of "marks" in memory that indicate whether the sign was 0 or 1. After 18 such operations, enough room is left in X1 for A6, and from there on, the rest is easy.

The difficulty of saving and restoring all the registers implies that a complete stack emulation is impractical; at least one register will have to "float." In practice, most of the 6000 software is coded with the convention that the caller of a subroutine is responsible for saving important register contents. A subroutine call will destroy any or all register values in general.

Of course, the machine status, including all register contents, is saved through the *exchange jump* (EXJ) instruction. However, this instruction can only be executed by a peripheral processor, and is only invoked when control of the CP is transferred to another process. EXJ is not available to a user.

10.6.10. Relocatable Linking and Partial Compilation

In Fortran, each subroutine in a set comprising a program is separately compilable in the 6000 system. In practice, small programs can be compiled so rapidly and cheaply that partial compilation is not often used. However, the capability exists and deserves some discussion.

The Fortran compiler arranges that every subroutine reside in a contiguous block of words (figure 10.23). The block begins with parameter space (if more than six parameters are needed), then a word for the RJ target, and then the subroutine code and data space. (Actually, a parameter count and the subroutine name are also present, for the sake of the loader.) Since all memory references are full 18-bit addresses, all the data can be collected at the end of the block, so that the instructions need not be interrupted by data.

The compiler prepares each subroutine block in the general form shown in figure 10.23, and generates code under the assumption that the block begins in location 0. It then prepares tables for the relocatable loader that specifies

1. The name of this subroutine.
2. The entry point location (target word for a calling RJ).
3. The location of every memory reference instruction (containing a K field) in the block. The location must include a position code specifying the position of the instruction in the word.
4. The name associated with every subroutine call and a pointer to the call. The RJ instructions associated with one subroutine are linked together. The system is similar to that in the HP-3000 system (figure 10.15).

The loader must adjust each of the K-reference instructions by the offset of the subroutine block within the program block. Also, every subroutine call (an RJ somewhere) must be fixed to point to the target subroutine.

Since partial addresses and indirect words are not needed, these loader operations are quite straight-forward and efficient. Of course, loading need only be done once after a compilation. The loaded program block can be saved and executed with no setup time at all.

10.6.11. A Pascal Implementation

Wirth [1971] describes an implementation of Pascal on the 6000 system. He found that several of the 6000 features can be exploited to yield an efficient implementation of this block-structured language supporting recursive and nested procedures.

Wirth's Pascal compiler generates absolute code (it uses the relocatable loader in a trivial way). The RJ instruction is never used, because it fails to support recursive procedure calls. The allocated memory program area is divided among the program, the stack, and the heap. Of course, these three comprise one program area to the operating system, with lower bound RA and extent FL.

The registers X1 to X5 are used as a stack to hold intermediate results during the evaluation of expressions. Wirth found that X1 was used 75% of the time, but X5 only 0.02% of the time, out of the 7927 compiled instructions that assign values to one of these registers. This means that very few expressions require more than 5 intermediate results. The Pascal compiler in fact declares a user error if more than 5 intermediate results are needed in some expression. Removal of this restriction through assignment of memory temporaries could be done, but would add to the compiler's complexity and yield a marginal benefit. Registers X6 and X7 are used to store results in operand fields in the stack or heap.

Registers B1 to B4 are used for the display (see chapter 9). These provide four nesting levels, of which only three are required for the Pascal compiler. Since new local variables can only be introduced by declaring a new procedure in Pascal, the compiler has at most three static procedure nesting levels.

The use of a B register as a display is especially attractive on the 6000. The set instruction form

```
SAi  Bj+K
```

is useful for fetching or storing local variables, and the form

```
SAi  Bj-K
```

is useful for fetching or storing formal parameter values. Parameters passed by reference, or pointers, can be accessed through an SAi Bj − K instruction, followed by an SAi Xi instruction.

Indexing through the display is achieved through the following sequence of instructions:

```
{evaluate index in register Xi}
SB5  Xi ± K  {K is the parameter
             offset from display}
SAi  Xi+Bk   {Bk is the display}
```

The last instruction replaces the index in Xi by the data value.

A procedure call is implemented essentially as discussed in chapter 9, except that Pascal supports neither call-by-name, nor branch label parameters. The necessary words to be written to the stack by the PE and BE operations are then only the dynamic link (procedure return address) and the static link. A 60-bit word can carry up to three address labels, so that a one-word stack marker is feasible.

Since the compiler generates absolute code, each of the procedure's addresses is known at compile time. The dynamic link can be written in two instructions as follows, where K is the return address and K′ + Bj is the stack location to which the address is to be written:

```
SX6  B0+K    {writes K to X6}
SA6  Bj+K′   {written to the stack}
```

The *powerset* type in Pascal provides a set membership facility for a finite set. The membership of some set element in a set S can be represented by a 1 or 0 bit in position c_i in a word assigned to that element. Then the machine's bit-parallel logical instructions ("and" for intersection, "or" for union, "not" for set inverse) can be used for set operations. A test for set membership is equivalent to a word shift followed by a sign-bit test. Wirth's compiler restricts the number of set elements to 60, but this is again a restriction that could be lifted through some additional work.

10.6.12. Summary

The major defects of the 6000 system for Pascal (or any other Algol-like language implementation, for that matter) are

1. The absence of a jump instruction that deposits the current processor status in a general register or in a register designated stack location.
2. The absence of logical shift operations. (All the shifts are arithmetic. They propagate the sign bit on a right shift, or left-circular on a left

shift.) A true logical shift would be desirable for the set membership test.

3. The use of one's complement arithmetic with two representations of zero. (Zero is either all zeroes or all ones).
4. The failure to indicate overflow in integer arithmetic and the different arithmetic magnitudes of the various integer instructions.
5. The absence of character-handling operations, needed in languages such as Snobol, Cobol, and RPG. Character-handling operations are supplied in the CDC Cybernet 7600 systems.

Apart from these defects, the 6000 system is among the most powerful and fast computer systems in existence and is unparalleled in the execution of large, floating-point, scientific programs. The simplicity and generality of its instruction set makes possible relatively simple compilers. However, optimization of the use of its register sets is a difficult problem, especially in this system, considering the memory access properties of registers A1 to A7.

Exercises

1. Describe in greater detail an implementation of PE, BE, EB and a procedure call, using the Pascal conventions and the 6000 instruction set. Assume that a packed stack marker is not necessary.
2. Develop systems for the allocation and access of local variables in a Pascal system, both static and dynamic.
3. Define the Fortran subroutine structure in more detail, and give a loader algorithm in Pascal.
4. Write a Compass (6000 assembler) subroutine that performs integer division, and yields the quotient and the remainder, for positive integers. (Access to a Compass manual and the 6400 reference manual would be helpful.)
5. Sketch an algorithm that lifts the restriction that an expression evaluation may invoke at most five temporary values.
6. Sketch an algorithm that lifts the restriction that at most five nesting levels are permitted. *Note:* Assume that B6 and B7 are needed as general purpose registers.
7. Examine the problem of saving and restoring all the registers. (Access to the complete instruction set would be helpful in this exercise. However, you may assume that the machine contains left and right arithmetic and circular shifts on the X registers.)
8. Given that only the field length FL may be altered during execution,

discuss the problem of dynamic allocation of memory for a Pascal compiler that requires a stack and a heap on this system.

10.7. The IBM System/360

The IBM System/360 is actually a set of compatible computer systems, covering a wide range of price and performance. The internal performance range between the largest and the smallest 360 system is approximately 50:1 for scientific computation and approximately 15:1 for commercial processing.

System compatibility is achieved through a common instruction set and virtual memory in all models. In the smaller models, some of the instructions are performed through software, and large programs generate considerable disk access.

Each system is based on a single CPU that has access to main memory and I/O channels. We will not be concerned with I/O access, as most of that is handled by the operating system, and may be set up with standard macros.

The CPU is time-shared between operating system and user operations, with operating system and I/O channel operations receiving higher priority in general.

Multiple CPU's can be interconnected and can access the same memory. Multiple CPU's can be used in a multiprogrammer system by assigning each one to a different task through a scheduler. The tasks may be part of one large program or separate user programs, but must function asynchronously.

A CPU contains 16 general purpose registers of 32 bits each, and four floating-point registers of 64 bits each. These are interconnected through system control to the main storage, which can consist of as much as 16.5 million bytes.

The 16 general registers are designated 0, 1, ... , 15, and the four floating-point registers are designated 0, 2, 4, 6. Although the same codes are used for the two sets of registers, different instructions are provided for loading and storing from the two different register sets and for operations on the register contents.

Long (64-bit) and short (32-bit) floating-point arithmetic is provided, with packing and unpacking operations. A full set of arithmetic operations on both forms is provided.

Integers may be fullword (32 bits) or halfword (16 bits). All fixed-point arithmetic is 32-bit twos complement, however. All 16-bit arithmetic is performed with 32-bit registers by setting carry and overflow indicators relative to the least significant 16 bits.

Instructions are also provided to process decimal data, in the form of packed 4-bit binary-coded-decimal characters, or zoned 8-bit characters. The

operands for decimal data reside in memory; the instructions carry addresses of the left-most operand bytes.

Data in memory is subject to alignment restrictions, as shown in figure 10.24. Main memory is byte organized; every address is a byte address. A halfword must be aligned on an even byte boundary, a fullword on an even halfword boundary, and a double word on an even fullword boundary. These restrictions apply to all the memory reference instructions. The alignment restrictions can only be broken through byte move instructions that can move a non-byte entity between an even and an odd location in memory.

A special 64-bit status register PSW carries several fields of machine status information, including the next instruction address, overflow, and carry indicators, and a *condition code*. The condition code is a 2-bit field in PSW that is set by various arithmetic operations and sensed by a set of conditional branch instructions. The code value 0, 1, 2, or 3 depends in a specific manner upon the instruction that affects it. Its interpretation through a conditional branch instruction is by a 4-bit field in the branch instruction; the bits correspond to codes 0, 1, 2, and 3, and the branch is taken if the bit corresponding to the current condition code is set. For example, if the condition code is 2, then any branch instruction bit pattern of the form XX1X causes a branch.

Program interrupts are possible as a result of certain operations, such as fixed- or floating-point overflow, or an address bounds violation. When a program interrupt occurs, the current instruction execution is suspended and an operating system service routine is called. Overflow interrupts may be

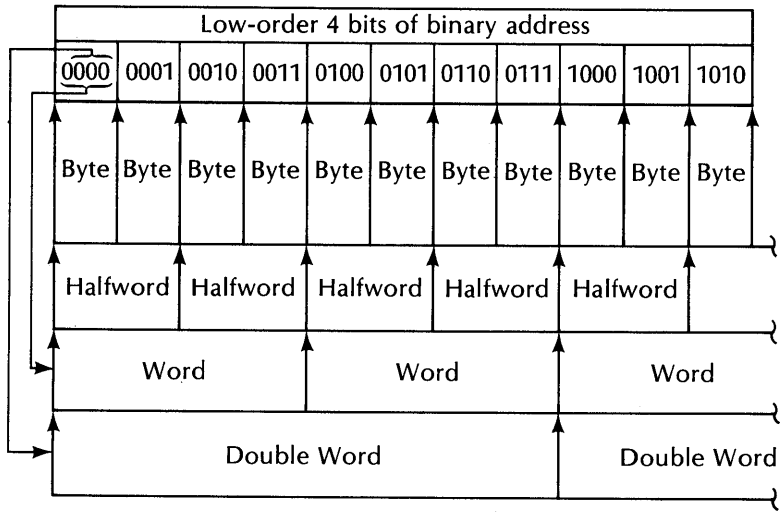


Figure 10.24. Data fields and alignment in memory for IBM 360.

brought under user control, if desired, through mechanisms that we shall not discuss.

10.7.1. Instructions

The five 360 instruction formats are illustrated in figure 10.25. Instructions are 1, 2, or 3 halfwords in length, depending on the op code, and must be aligned on halfword boundaries in memory. Memory is shared between instructions and data in any desired manner.

An operation code occupies the first 8 bits of every instruction. The remainder of the instruction designates one or more registers (R, B, X), and possibly one or two address displacements (D). The SI format carries an 8-bit immediate operand (I2). The SS format carries one or two length operands, L or (L1, L2). The SS format is used for memory operands, e.g., decimal arithmetic, byte moves, and byte string comparisons.

A "0" as an index or base register designation means that no register is used in that part of an address calculation. However, register 0 can carry a data value and be accessed by instructions that refer to the register's contents as data rather than as an address component.

Scalar nondecimal operations are between two registers, or between a register and a memory location. The RR instruction format (figure 10.25) always specifies a register-to-register binary operation. Thus

```
OP   R1,R2
```

for a binary operation OP is equivalent to

$$C(R1) \leftarrow C(R1)\ \text{OP}\ C(R2)$$

The RX format specifies a register-to-memory operation in general; it is written symbolically as

```
OP   R1,D2(X2,B2)
```

The first operand is C(R1). The second operand is

$$C(D2+C(X2)+C(B2))$$

Here, $C(X2)=0$ if $X2=0$, and $C(B2)=0$ if $B2=0$. That is, the contents of X2 and B2 are added to the displacement D2 to form an absolute memory address of the second operand. Although the functions of X2 and B2 in this instruction are interchangeable, X2 is usually considered the *index* register, in the sense that it might be used to access an array element, and B2 is

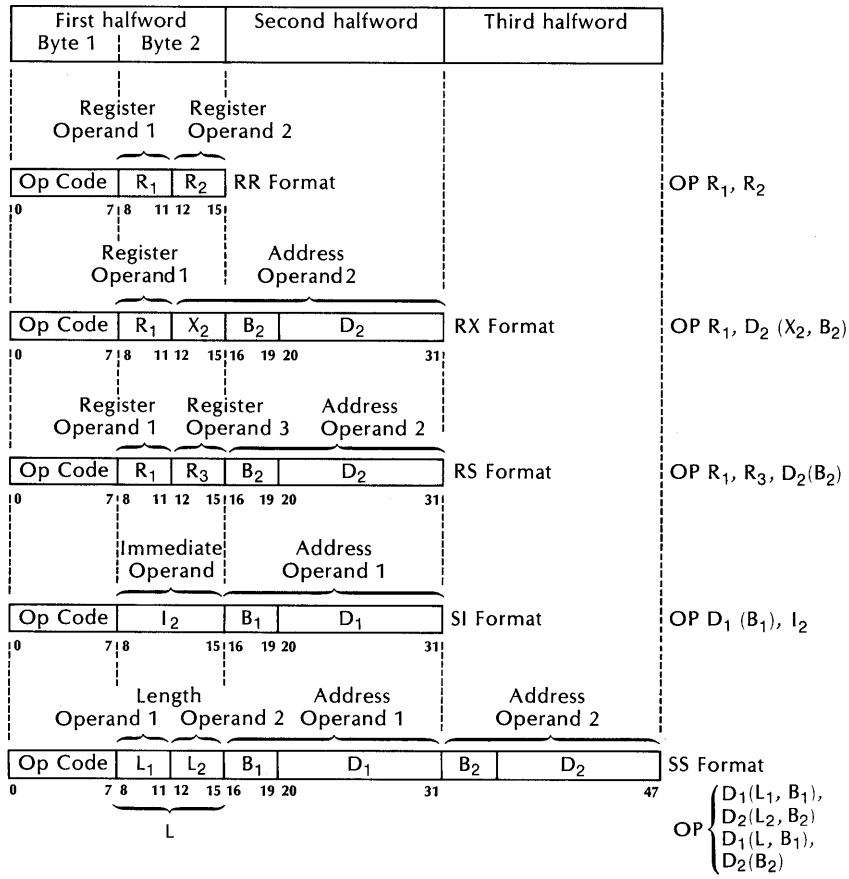


Figure 10.25. The five basic instruction formats for IBM 360.

considered a *base* register. The base register is needed to specify the loading origin of the program and cannot be used as a second index register.

The displacement D2 provides relative addressing up to 4095 bytes (the displacement field is 12 bits) beyond the base and index register address. Each of the three address components are treated as positive numbers. However, address arithmetic is performed without regard to overflow, so that the effect of negative indexing is achieved by regarding the index register as a positive or negative twos-complement 24-bit number. For example, if C(B2) = 43000, D2 = 1600, and C(X2) carries the (hexadecimal) value FFFFFFFC the effective address is 43000 + 1600 − 4, since C(X2) represents −4 in twos complement form.

The effective address must lie within a memory partition assigned to a user,

or else an interrupt will occur. This provision protects one task in memory against an unauthorized access of its memory by another task. Only certain operating system procedures have unrestricted access to all the system's memory. Protection is achieved by dividing main storage into pages of 2048 bytes each, and by associating a 5-bit key with each page. A user's right of access to storage is identified by a 4-bit protection key. Access to some storage location is only granted when the user's protection key matches the storage page key. A violation causes the instruction to be suppressed or terminated and execution to be altered by an interrupt.

When a task is initiated or resumed, the memory protection keys are all set to delimit a section of memory for the task; usually, the task is permitted to access only one contiguous partition of memory for reading or writing of data. The memory partition assigned to the task is determined when the task is initialized and is frozen in location until it is completed. However, the partition may be rolled out and later rolled in asynchronously by the operating system for the sake of overall system scheduling needs.

10.7.2. An instruction sample

A few of the 360 instructions are described below. A complete instruction set may be found in any of several references and in the IBM System/360 Principles of Operation manual.

Register Load

```
LR   R1, R2          Load
L    R1,D2(X2,B2)
```

The second operand is placed in the first operand location. The second operand and the condition code are unchanged.

```
LH   R1,D2(X2,B2)    Load Halfword
```

The halfword second operand is placed in the first operand location. The second operand is expanded to a full-word by propagating the sign-bit value through the upper 16-bit field.

The LH instruction copies a halfword (16 bits) in memory specified by D2, X2, B2 to register R1. The halfword operand is expanded to form a full 32-bit twos-complement arithmetic operand by propagating the sign bit into the 16 high-order bit positions. The condition code is unaffected, but an addressing interrupt may occur.

```
LTR  R1,R2      Load and Test
```

The second operand is placed in the first operand location, and the sign and magnitude of the second operand determine the condition code. The second operand is not changed. The resulting condition code is: If result is zero, then 0. If result is <0 then 1. If result is >0 then 2.

The LTR instruction is identical to the LR instruction, except that the condition code is set according to the sign and magnitude of the operand, as indicated in the figure.

Other LOAD instructions

Twenty-two LOAD instructions are provided in all, for all the fixed- and floating-point data types provided in the system. Some provide negation or complementation during the copy. Thus the LPSW, load program status word) instruction loads the system's 32-bit status word PSW into a selected register. The LA instruction loads the 32-bit address of the operand into the designated register. The LM instruction loads a group of the general registers from a contiguous group of memory locations. The LM instruction and its complement STM facilitate saving and restoring the general registers. One byte can be loaded or stored through the IC or STC instructions respectively. IC is not a true load instruction, as it only changes the least significant byte in the register; the most significant bytes are unaffected.

No LOAD instructions are provided for decimal operands, inasmuch as these have variable lengths and are operated upon directly in memory by SS instructions. Of course, parts of a decimal operand can be loaded or stored through the byte and word load/store instructions.

Fixed- and Floating-point Arithmetic

```
AR   R1,R2      Add
A    R1,D2(X2,B2)
```

The second operand is added to the first operand, and the sum is placed in the first operand location. Addition is 32-bit 2's complement. The condition code is set: if sum is zero then CC = 0; if sum is <0 then CC = 1; if sum is >0 then CC = 2; if overflow then CC = 3.

The AR and A instructions are typical of the arithmetic instructions: subtraction, multiplication and division of all the fixed and floating-point data have similar instruction forms.

Comparisons

```
CR   R1,R2      Compare
C    R1,D2(X2,B2)
```

The first operand is compared with the second operand, and the result determines the setting of the condition code. Operands are regarded as 32-bit signed 2's complement integers. If the operands are equal then CC = 0; if first operand is low then CC = 1; if first operand is high then CC = 2.

The compare instructions CR and C specify two operands, but only set the condition code. The operands are unaffected. An addressing interruption is possible on C.

Branches

```
BCR   M1,R2        Branch on Condition
BC    M1,D2(X2,B2)
```

The updated instruction address is replaced by the branch address if the state of the condition code is as specified by M1; otherwise, normal instruction sequencing proceeds. The branch address is specified by the second operand R2, or D2+(X2)+(B2). The M1 field is used as a four-bit mask; each bit corresponds, from left to right, to one of the four possible condition codes. When all four M1 bits are set, the branch becomes unconditional; when all four are clear, the instruction is a no-operation.

For example, if the current condition code is 2 and the mask carries the binary value 0010, then the condition is satisfied and the branch occurs. The branch address is either in register R2, for BCR, or is specified by the usual set (D2, X2, B2), for BC.

A program interruption will occur on the next instruction, if its location is outside the range of accessible memory.

Subroutine Call

```
BALR  R1,R2        Branch and Link
BAL   R1,D2(X2,B2)
```

The rightmost 32 bits of the PSW (program status word), including the updated instruction address, are stored as link information in the general register specified by R1. Subsequently, the instruction address is replaced by the branch address. If the R2 field contains zero, the link information is stored without branching.

These are useful for subroutine calls and for setting a program address in a base register. The right-most 32 bits of the PSW will carry the byte address of the next instruction (following BALR or BAL) when it is loaded, hence register R1 will be loaded with a "return address". The branch target is specified by the contents of register R2, for BALR, or the set (D2, X2, R2), for BAL. As usual, an address interrupt will occur on the next instruction if it lies outside accessible memory.

Other Instructions

The few instructions defined above should provide some flavor of the System/360 architecture. There are 142 instructions altogether. These cover arithmetic, comparisons and conversions of all the data types. There are in addition logical operations (OR, AND, XOR, NOT, shifts) on the fixed-point types, several kinds of moves, some I/O, and some special edit instructions to facilitate business report writing.

Since our concern is primarily with register and addressing conventions, we shall not define more instructions.

10.7.3. Procedures and program relocation

Procedures may be independently compiled and later linked together. A compiler prepares a file called an *object module* for each procedure. A set of object modules are then linked by a linking loader to form a *load module*. A load module can then be assigned and written to a region of memory just before execution by the absolute loader. A load module may require a few adjustments, as we shall see, or not, depending on the source program requirements.

Two kinds of relocation will occur in general, and they are handled by the *linking loader* and the *absolute loader*.

The *linking loader* brings together several object modules into a single load module. In general, there will be many subroutine call instructions and memory addresses that must be adjusted to conform to the locations assigned to the subroutines comprising the module. The linking loader cannot necessarily make all the adjustments needed for relocation, and therefore must prepare a special relocation table for the absolute loader.

The *absolute loader* accepts a load module and positions it somewhere in memory. It must adjust certain address constants to reflect the memory position to which the program has been assigned; these constants are listed in a table prepared for it by the linking loader.

Relocation is achieved in two ways—through a *base register* convention, and through adjustments to *address constants*. A base register is a designated register (usually number 13) that contains the absolute loading address of the program in memory. Every memory reference instruction must contain a base register component, so that every reference will be correctly made during execution. Failure to specify a base register on some instruction or failure to establish or maintain the relocation in a base register will cause a bounds violation or other program failure during execution.

An *address constant* is some memory fullword that contains the absolute program loading location as a component. Such addresses must be adjusted by the absolute loader just prior to execution. As we shall see, address constants are an important part of 360 run-time structures.

10.7.4. Save Areas

Every program module that will pass control to some operating system control program and back again must provide a *save area*. A save area is comparable to the stack marker in the AOC machine and is a contiguous block of 18 fullwords. It "belongs" to the calling program, not to the callee, yet the callee uses it to save registers. Figure 10.26 shows a save area belonging to a procedure Y, called by procedure X. When Y calls a procedure Z, Z fills Y's save area.

The save area contains 15 fullwords for the general registers 0 to 12, 14 to 15, a fullword for the address of the previous save area, a fullword for the address of the next save area, and a fullword used by PL/I.

The address of a caller's save area is passed in register 13. The callee then bears the responsibility for filling the save area, and restoring the register contents upon an exit. The register set 0 to 12, 14 to 15 can be saved (restored) by a single STM (LM) instruction.

Recursive or reenterable programs require allocation of a new save area upon each call and its deallocation upon return. Otherwise the save area may be a fixed component of a program data space.

The forward and backward chains through save areas facilitates debugging and abort analysis; the chain provides useful information regarding the procedure call nesting, the entry and exit parameters, and return addresses.

Register 14 is assumed to carry the return address upon a procedure call. Register 15 is assumed to carry the entry point address of the called

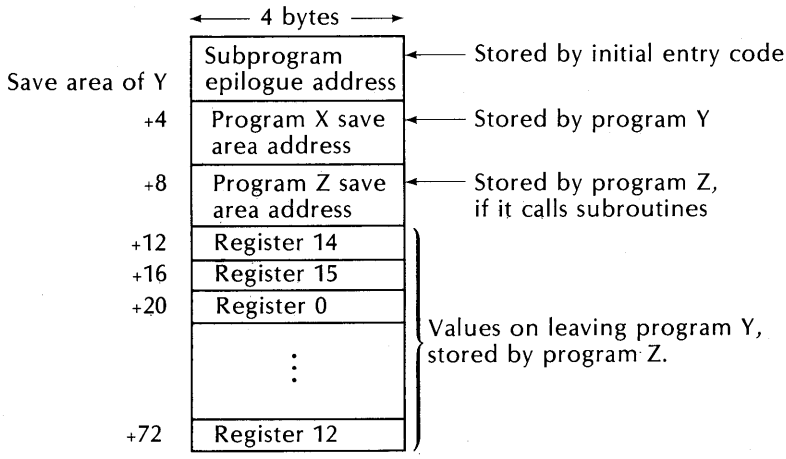


Figure 10.26. A save area.

procedure. Registers 0 and 1 are generally used to pass the address of a parameter list to the called program. These and register 13 are called *linkage registers*. Although the machine architecture requires no special register use conventions, the operating system control programs, debug and dump systems, and the system macros do depend on these conventions.

Upon a return from a procedure, the registers must in general be restored to their state upon entry. However, register 15 can be used to return a *return code*, useful in indicating any unusual conditions or errors found during execution of the procedure. A return code of zero is interpreted by the system as "no errors—nothing unusual". Any other code can be used, but of course should be documented. Registers 0 and 1 need not be restored; recall that they were used to pass parameter information to the procedure. Register 14 contained the return address upon entering the procedure, so the return is effected by a BR 14, executed after restoring register 14.

10.7.5. Object Modules

A compiler prepares an object module for each procedure in general. An object module consists of three tables—ESD (External Symbol Dictionary), TXT (Text), and RLD (Relocation and Linkage Directory).

The TXT table is a list of fullwords to be initially written to memory. Instructions, constants, initialized, and uninitialized data are specified through a TXT table. Some of these fullwords must later be modifed by the linking loader and by the absolute loader. An initial loading address of 0 is assumed by the compiler.

The ESD table contains every symbol defined in this segment that must be referenced externally, and every symbol defined elsewhere that is referenced in this module. Three kinds of ESD entry are needed as follows:

Symbol	Type	Location	Meaning
RESULT	SD	56	Module definition
SUM	ER	864	External reference
COMP	LD	270	Local definition

RESULT is a name given to the module being constructed, and it is useful as an entry point location definition (in this case, the first executable location is 56) and as a name for use in a symbolic loader map. The 360 loader will support multiple entry points into a given module, so there could be several of these.

The address of SUM is needed in location 864, however SUM is externally defined. The loader must therefore determine the address of SUM and fill location 864 with it. The ER type may also be used for calls of external procedures; a procedure is called on the 360 by loading register 14 with its

address (from memory, where it was set up by an ER entry in the ESD table), then executing BR 14 (actually BCR 0,14).

COMP is defined locally, but its address is needed externally.

The RLD table is used exclusively for address constants appearing in the program segment. Recall that an address constant is some memory word whose value depends on the location of this program or some external program. For example,

```
DC    A(PG1 - PBXX + SAM -14 +FRED +17)
```

might appear in an assembly language program. The mnemonic DC means *define constant*, and A(. . .) means "address of (. . .)". Here, the constants 14 and 17 would be combined by the assembler. This constant would then appear as a "3" in some location, say 374. Then four RLD entries, one each for PG1, PBXX, SAM and FRED, would be generated, all referring to location 374.

The RLD table contains the following information:

- The location and length of each address constant that must be modified.
- The external symbol by which the address constant must be modified.
- The modification (addition or subtraction).

Several RLD entries may refer to the same address constant. Each of them causes some modification to the constant—an addition or subtraction of an externally defined address. Thus the above DC would result in a DC 3 in the TXT table, and four RLD table entries for PG1, PBXX, SAM, and FRED.

An address constant as seen by the absolute loader will then be one of the following:

1. *Absolute.* Will occur if addresses in an address expression appear in pairs, one added, the other subtracted. The absolute loading location of the load module will therefore not affect the address constant.
2. *Simple relocatable.* Will occur if all but one of the address expressions appear in additive-subtractive pairs, and the one left over is an additive address. The absolute loading address of the load module must be added to the address constant by the absolute loader.
3. *Compound relocatable.* Will occur if the absolute loading address of the load module must be added to the address constant twice or more, or subtracted once or more times.

Clearly, the absolute loader need not be informed of case 1 at all—the linking loader was able to resolve the address constant completely. Case 2 is very common, and requires one RLD entry. Case 3 is likely to occur only in unusual assembly language programs; it requires more than one RLD entry.

10.7.6. Addressing

The 360 memory reference instruction carries a base register, an index register and a displacement (see figure 10.25). Now the displacement field is relatively small, 0 to 4095 bytes. If a procedure carries more than 4096 bytes of data, then parts of that data cannot be accessed through a single fixed base register alone.

A solution to data addressing, when more than 4096 bytes of data must be accessed, is illustrated in figure 10.27. Here, data resides in some contiguous block of memory. Simple variables are grouped together, followed by arrays and other indexed structures. Another section of memory is dedicated to a *base table*, which contains address constants that point into various regions of

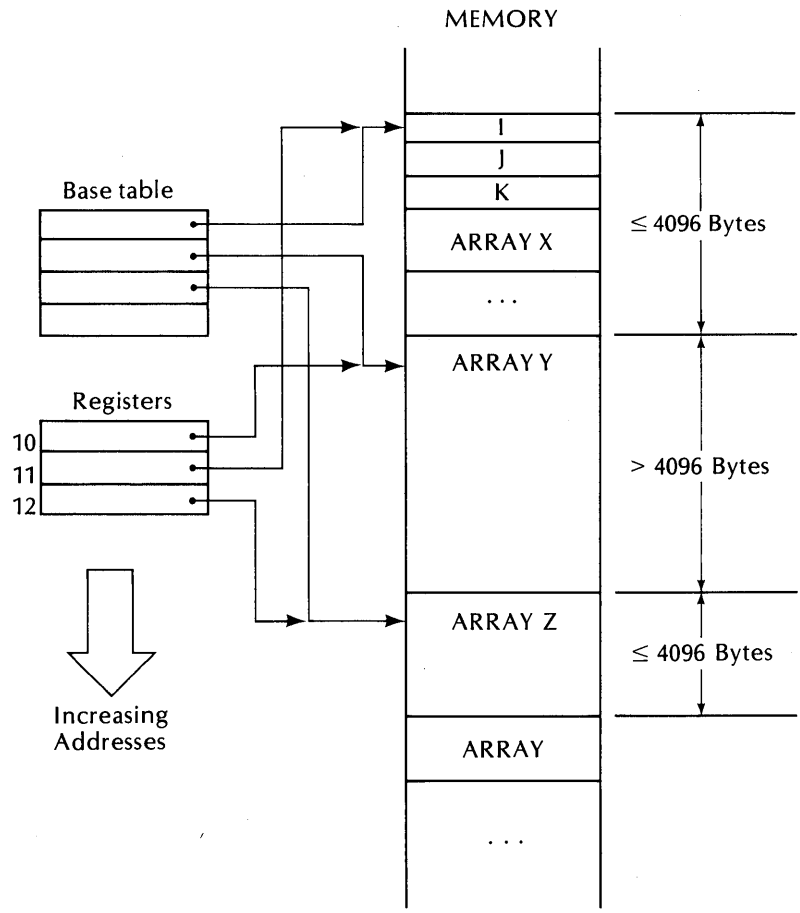


Figure 10.27. Base table and use of base registers to access data.

data memory. For example, the first base table entry might be a pointer to the first data cell. This base address, when placed in a register, can then serve to access any data element or array whose first location is less than 4096 bytes from the base address. Thus variables I, J, K, and all of array X (X can be any size) are accessible through the first base table address.

Subsequent base table entries are addresses of data memory farther than 4095 bytes from the previous one. Thus, if array Y is the first data area located more than 4095 bytes from variable I, then a base table element must be allocated to point to it. Here, Y can again be of any size, since a 32-bit register is available for indexing from the base table address.

For example, if registers 10, 11, and 12 are available as data base registers, then at any one time, each of these will in general point somewhere into data memory. If some data memory element is needed that cannot be reached from one of the registers 10 to 12, then one of these must first be replaced by a base table element.

The optimum assignment of registers to base addresses is clearly an interesting and difficult problem. For a very small program, some or all of the variable values themselves may be carried in registers, obviating many memory references. A larger (but not too much larger) program would permit one data base register through which all data accesses are possible. A still larger program requires a base table and dynamic setting of the base registers from the base table.

10.7.7. Object Module Design

It is best to establish an overall design of the object module and conventions for the use of the registers before anything else in a compiler is designed. We shall outline the design of the Fortran IV (G) compiler released by IBM in the late 1960's. We shall not describe the parsing and other algorithms in detail, but will look at some of the conventions of data allocation and code generation used in this system.

The arrangement of the program module, as generated by an absolute loader for one Fortran subroutine, is given below.

Heading (entry point)
Save area
Base table
Branch table
EQUIVALENCE variables
Scalar variables
Arrays
Subprogram argument lists
Subprogram addresses
NAMELIST tables

Literal constants (except those used in
DATA and PAUSE statements)
FORMAT statements
Temporary storage and constants
Program text

To a compiler, the program module is a TXT file. It must be modified by the linking and absolute loaders prior to execution.

Control passes into the first word of the program module. Some preamble instructions are in the *heading*. Among other things, the heading code sets general register 13 and saves all the registers.

The save area and the base table have been described above. The base table will contain absolute addresses during execution, and must therefore be adjusted by both loaders prior to execution.

The *branch table* contains absolute branch addresses, one for each branch target. Each branch target is assigned a fullword in the branch table, and a branch is then executed by loading a register R from this table, then executing BCR 0,R. The use of a branch table facilitates compilation in one pass without fixup; the branch instructions may be emitted in final form as they are encountered, and the branch targets may be entered in the branch table in suitable form as they are encountered.

The next three tables (equivalence variables, scalar variables, and arrays) carry the subroutine data. The equivalence variables are not dimensioned. An average Fortran subroutine will not have too many simple variables and nondimensioned equivalenced variables, so that these two tables can usually be accessed through one base register. The array table may require more base registers, as suggested by figure 10.27.

The next table (subprogram argument lists) carries the addresses of the arguments for all subprograms called by this subroutine. If N different subroutines are called, then there are N lists in this table, one list for each subroutine. Upon calling a subroutine, its address list is first prepared, then the address of its address list is passed in general register 1. The called subroutine can then access its parameters indirectly through register 1.

The next table (subprogram addresses) contains one entry for each subroutine called by this module; the entry holds the subroutine address, as supplied by the linking loader.

The *namelist* table is used for the NAMELIST feature provided in this Fortran version. NAMELIST is a Fortran feature for the symbolic access of program input/output file data.

Literal constants are held in the next table. These are sharable; if a "20" appears as an arithmetic constant in several places in the subroutine, it only appears once in this table.

The FORMAT statements appearing in the subroutine are coded in a

condensed form and placed in the next table. FORMAT is used by a run-time program to control the form of input and output, and its conversion to/from internal form. The run-time program interprets a list of characters in the FORMAT table.

The last data table consists of memory cells allocated for intermediate results and other constants. Registers are used as much as possible for intermediate results, for obvious efficiency reasons. However, only four registers are available as accumulators for intermediate values, because of other register uses. This data area provides intermediate result storage for any surplus beyond four values.

The program instructions occupies the last program module area. The first entries are occupied by statement functions, if any, with a branch around these to the first program instruction.

10.7.8. Register Allocation

The four floating-point registers are used as needed as floating-point accumulators. The sixteen general registers are allocated as follows:

Registers 0, 2, 3 are used as accumulators. Recall that register 0 cannot be used as an address register in an instruction. These registers are saved in a subroutine call and can therefore hold intermediate results preserved in a call.

Register 1 is used as an accumulator within a subroutine, but also to pass the address of the argument list when calling a subroutine.

Registers 4 to 7 contain index values as needed for reference to array variables, where the subscripts are linear functions of DO variables and the array does not have variable dimensions. The Fortran IV (G) compiler provides some optimizations for array indexing in DO statements; an array index that is incremented by a constant amount on each iteration of the DO loop is carried in one of these registers and incremented by the amount on each iteration with one instruction.

Registers 8 and 9 contain index values as required for references to array variables, where the subscripts are of the form $x \pm c$, where x is a non DO-controlled variable and c is a constant.

Register 9 contains index values as needed for nonlinear subscripts, not covered by the one of the above cases. The motive for distinguishing these three cases appears to be to subdivide the array indexing-DO loop cases for optimization.

Registers 10 to 12 contain base addresses loaded from the base table (see figure 10.27). If the subroutine has more than three base table entries, these registers must be dynamically changed, otherwise they may be initially loaded from the base table and never changed.

Register 13 contains the address of the object module save area, following the system conventions outlined previously. It is also useful as a base table

address reference, since the base table follows the save area.

Register 14 contains the return address for subprograms and the address of branch target instructions during the execution of branches.

Register 15 contains the entry point address for subprograms as they are called by this program module.

10.7.9. Summary

The 360 system architecture is based on a set of 16 general purpose registers and a set of register-register or register-memory data operations. A typical memory reference instruction carries a displacement and two register references. The displacement is limited to 4095 bytes, one of the registers depends on the absolute program loading address fixed just prior to execution, and the remaining register can be used for array indexing.

A compiler generates an *object module*, consisting of a TXT file containing instructions, constants and partially fixed-up addresses, an ESD table that specifies symbolic information for linkage to other object modules, and an RLD table that specifies how addresses are to be fixed up by the loaders. The nature of the machine is such that a program module must remain in the memory partition to which it was assigned throughout its execution.

For maximum run-time efficiency, the allocation of registers as array indices, base registers and data registers, and their dynamic assignment as such is important. Data access usually requires a *base address table* containing addresses into selected locations in data memory. The base addresses are separated by at least 4096 bytes, since any location within that range can be accessed through the base address and a displacement in the instruction.

Subroutine calls are governed by system conventions in which every subroutine must provide a save area, whose address is passed in register 13. The called subroutine is expected to save all but register 13 (but none of the floating-point registers), and restore these upon return. Other save area entries facilitate debugging and the tracing of subroutine calls upon an abnormal program termination. The *save area* system can be adapted to recursive procedure calls by dynamically allocating a new save area on each call and deallocating it upon a return.

The 360 system is probably the most widely used computer system in the world, despite certain shortcomings as a time-shared multiprogramming system. Its architecture has also been copied in non-IBM machines (e.g., Interdata and Amdahl), in order to exploit 360 software. It will undoubtedly continue to be among the major computer systems in the world for many years.

Exercises

1. How suitable is the IBM 360 architecture as an AOC machine simulator? Sketch a plan for support of a stack, variable addressing and procedure calls that exploit the machine's registers and addressing modes. Your system must be relocatable, or it will not be executable on a time-sharing 360.
2. When a 360 program begins execution, it is assigned to a memory partition that cannot be changed during its execution. Why not? What conventions must be satisfied by every program in order that the system could move programs about in memory arbitrarily during execution?
3. A save area is part of the caller's code, but the callee fills the save area to save registers, and later accesses it to restore the registers before returning. Is this a reasonable plan? Why or why not?
4. Compare the object module system for the IBM/360 with the USL structure for the HP-3000. What differences are dictated by the architectural differences? In what ways are the capabilities different?
5. Sketch an algorithm for the assignment of a limited number of registers as memory data pointers (see figure 10.27 and the associated discussion) and their management during execution. Strive for a reasonably efficient run-time implementation.
6. Sketch a Pascal system similar to Wirth's scheme for the CDC 6400 that exploits the 360 architecture. Impose whatever reasonable limitations on procedure nesting depth or expression temporaries, etc., that seem reasonable; however, avoid any limitation on the maximum size of a program if possible.
7. What changes would you make in the 360 architecture that would better suit the machine for time-sharing purposes? For the support of block-structured languages? For the support of Fortran?

10.8. A Generalized Code Generator

As the preceding machine examples indicate, the generation of code for some target machine by a compiler can require highly specialized algorithms tailored to the particular target machine. A number of alternative approaches to compiler construction have been proposed that attempt either to generalize code generation or to isolate the machine peculiarities in some fashion, in order to reduce the effort required to implement languages on more than one target machine.

A promising approach to more abstract code generation has recently been reported by Glanville and Graham (Glanville [1978]). It does not deal with all the issues of machine dependence and machine idioms, but does represent a significant contribution to the problem of selecting a suitable sequence of object code instructions, given an abstract syntax tree. The method has the virtues of generality and correctness. The target machine instruction set is represented as a table, and if the table is correct, the generator will either block or generate correct code.

The method requires an abstract syntax tree (AST) and a syntax-directed translation scheme (SDTS) as input. The SDTS is a description of the machine's instruction set and the arithmetic operations expressed in prefix notation. The AST is first translated to prefix by scanning its nodes in preorder. The prefix form of the AST may then be parsed by a modified LR parser, to yield a sequence of SDTS production numbers; the output sequence then represents the desired object code instruction sequence.

The SDTS underlying grammar will in general be ambiguous. This means that a deterministic LR parser can only be constructed by arbitrarily resolving its inadequate states in favor of one reduction or another, or a push action. It is essential that an arbitrary resolution of an inadequate state not result in a machine block on some input string. We shall develop necessary conditions on the SDTS that the machine not block.

The resolution of an inadequate state that provides two or more reduction possibilities should in general be toward the largest of the possible productions. As a result, the system selects instructions that encode the largest parts of the AST and therefore minimizes the number of generated instructions. Although the method cannot be proven to yield optimal code, some experiments have shown that the quality of its generated code is very good, with no attempt to improve the code quality by any special means.

The method does not directly deal with questions of register allocation, base register movement, etc. These matters should be handled with auxiliary algorithms that can interact with or accept code generated by this generalized system.

Example

A small set of machine instructions that have the general form of an SDTS grammar (chapter 7) is given below.

```
1. r.2 ::= (+↑+k.1 r.1 r.2)     "add r.2,k.1,r.1";
2. r.1 ::= (+r.1↑+k.1 r.2)      "add r.1,k.1,r.2";
3. r.1 ::= (+↑k.1 r.1)          "add r.1,k.1";
4. r.1 ::= (+r.1↑k.1)           "add r.1,k.1";
5. r.1 ::= (+r.1 r.2)           "add r.1,r.2";
6. r.2 ::= (+r.1 r.2)           "add r.2,r.1";
```

```
 7. λ ::= (:=↑+k.1 r.1 r.2)     "store r.2,*k.1,r.1";
 8. λ ::= (:=+k.1 r.1 r.2)      "store r.2, k.1, r.1";
 9. λ ::= (:=↑k.1 r.1)          "store r.1,*k.1";
10. λ ::= (:=k.1 r.1)           "store r.1,k.1";
11. λ ::= (:=r.1 r.2)           "store r.2,r.1";
12. r.2 ::= (↑+k.1 r.1)         "load r.2,k.1,r.1";
13. r.2 ::= (+k.1 r.1)          "load r.1,=k.1,r.1";
14. r.2 ::= (+r.1 k.1)          "load r.2,=k.1,r.1";
15. r.2 ::= (↑r.1)              "load r.2,*r.1";
16. r.1 ::= (↑k.1)              "load r.1,k.1";
17. r.1 ::= (k.1)               "load r.1,=k.1";
```

The symbol to the left of "::=" represents the destination of the result of a unit computation, and the prefix expression to its right describes the computation. The parentheses are metasymbols introduced for readability. Symbol λ indicates that there is no resulting register value; the instruction is executed solely for its side effects.

Symbol "r" represents a general purpose register, and "k" denotes a literal constant, typically an address offset. The store operator is ":=" and the contents operator is ↑. The symbols r.1, r.2, k.1, etc. are SDTS nonterminals. If r.1 appears in the left member of a translation rule, then it corresponds to r.1 in the right member. Symbols r.1 and r.2 then possibly represent distinct registers. Thus the first ADD instruction (rule 1) adds the value in the memory location designated by k.1 and the contents of register r.1, leaving the result in register r.2.

Commutativity of operations is explicitly indicated by separate translation rules, e.g., rules 3 and 4.

Semantic restrictions can be imposed during the parsing process on instruction descriptions, permitting a greater number of special instructions to be incorporated in the translation rules. A single syntactic instruction pattern may then correspond to more than one instruction sequence. Thus two identical patterns (e.g., add r.1,r.2 and add r.2,r.1) could appear because of commutativity. Also one instruction could be a special case of another (e.g., "inc r.1" and "add r.1,k.1").

The source language must first be reduced to an intermediate representation (IR) that is a sentence in the underlying grammar of the instruction set SDTS. The IR may be generated from an AST by scanning it in preorder to yield a prefix expression. For example, the statement

$$A:=B+C$$

might translate to the IR expression

$$:= + k.a\ r.7 + \uparrow + k.b \uparrow r.7 \uparrow k.c$$

where a,b,c designate constants and r.7 is a local base register. In this

translation, we have obviously assigned memory to the variables A, B, and C, so that the AST translation will appear in register notation.

The translation of an IR sentence to a sequence of machine instructions is carried out by an LR(1)-like deterministic parser. The parser is constructed from the target machine description, replacing λ by X and adding rules $S \rightarrow Z$, $Z \rightarrow ZX \mid \epsilon$ to the grammar.

The code-emitting semantics implied by the SDTS translation rules are handled essentially as described in chapter 7. For example, when a shift is performed with the input symbol r standing for a register, the specific register represented by that r is pushed onto the semantic stack. Then when a reduce is done, the semantic information necessary to generate a final machine instruction is available in the semantics stack.

The instruction to be used when a reduce action contains more than one instruction (inadequate state) is picked by a simple heuristic. Instructions are ordered by the table constructor into a "best instruction first" sequence. At code generation time, the instructions in the set for a specific reduce action are tested in that order until one is found that is semantically compatible with the information in the semantics stack. Since all instructions in a reduce set have the same instruction pattern, the cheapest instructions (according to some cost criteria) are tested first. It is possible for some instruction in the reduce action state to not be selected, owing to semantics restrictions. For example, an "add immediate" could be chosen if the immediate constant is sufficiently small, otherwise, a more general (and more costly) instruction must be selected.

Register allocation must occur as a subtask to the parsing action. After an instruction pattern has been semantically verified, and the result is non-λ, and if the result register is not semantically linked to some other register in the instruction pattern, then the register allocator will be requested to provide a free register of the appropriate class, marking it "used". If the result register r is semantically linked to a register in the instruction pattern, then r must be used as the target register and marked "used". Recall that the result register is always one of the operand registers in the IBM 360 architecture, but need not be in the CDC 6000 architecture.

As an example of a generated code sequence, the input expression

$$:= + \text{k.a r.7} + \uparrow + \text{k.b} \uparrow \text{r.7} \uparrow \text{k.c}$$

yields the code sequence

```
LOAD  r1,r7
LOAD  r2,c
ADD   r2,b,r1
STORE r2,a,r7
```

using the SDTS given above. (This example is worked out in detail in Glanville's paper.)

10.8.1. Table Construction

The table constructor first treats the underlying grammar as a context-free grammar and constructs the set of LR(0) states, ignoring inadequate states for the moment. There will be numerous inadequate states in general, and they will arise both from finite lookahead and from grammar ambiguity. The construction algorithm seeks to resolve these conflicts in favor of short instruction sequences—shift-reduce conflicts are resolved in favor of shifting. For the most part, more powerful instructions are used as more of an expression is shifted into the stack before an instruction covers it. Reduce-reduce conflicts are often resolved by the semantic restrictions; if not, the longest instruction is used. The parser clearly must be capable of identifying reduce-reduce conflicts during parsing and refer their resolution to a semantic resolver. In resolving shift-reduce conflicts, SLR(1) lookahead information is used to ensure that reduce actions are included where needed.

Given a general context-free grammar, the conflict-resolution rules would not necessarily yield a recognizer for the entire language generated by the grammar. However, the construction is language-preserving for uniform instruction sets, defined as follows.

Uniform Instruction Set

Let Σ designate the vocabulary of the instruction set description. An instruction set is said to be *uniform* if it satisfies the condition:

Any left (right) operand of a binary operator b is a valid left (right) operand of b in any prefix expression of Σ^* containing b. Any operand of a unary operator u is a valid operand of u in any prefix expression of Σ^* containing u. An instruction set is uniform if its description is uniform.

The essential idea of uniformity is that the operands of an operator must be valid independent of their context.

For example, consider the SDTS given above. The operands of := are all either registers (rules 10, 11) or special cases of registers (rules 7, 8, 9). The operands of the left-most + in rules 1 to 6, 13, 14 are all either registers (rules 5, 6) or prefix expressions that become registers by rules 12, 16, or 17. Since a register is an argument of $\uparrow$ (by rule 15) and of + (by rule 5), the arguments of the other occurrences of + and $\uparrow$ represent special cases (i.e., ambiguities). Thus whenever :=, +, and $\uparrow$ occur, their operands are any prefix expressions corresponding to registers.

An example of an instruction set that is not uniform is

```
r ::= (+ ↑ k k)
r ::= (+ k ↑ k)
λ ::= (:= r r)
```

Here, k is a valid first operand of + only if the second operand is $\uparrow$k and vice-versa.

Glanville shows how the uniformity condition can be verified by the parser constructor algorithm.

Given uniformity, an SLR(1) code generator can be constructed that can be shown to fail to reach its ACCEPT state in only one of the following two cases:

1. The code generator loops infinitely.
2. The input IR is not syntactically within the sequence of prefix expressions described by the instruction set.

Looping can occur through a recursive sequence of single production reductions, e.g., $A \Rightarrow B \Rightarrow C \Rightarrow \ldots \Rightarrow A$, which can occur in practice if simple register-to-register moves are permitted. The potential of looping is easily tested in the finite state control and can be broken by a kind of "state-splitting" of certain of the generated states.

We need to consider the possibility of blocking—which would correspond to a syntax error if the input source were not generated internally from an AST.

The reduce action generates code by checking the semantic qualifications of the matched instruction pattern against the semantic qualifications indicated on the stack. If the set of instructions associated with a given reduce action are all semantically constrained, it is possible that none of the instructions will be compatible with the semantics on the stack. In this case, the code generator is said to *semantically block*. Semantic blocking can be avoided by the use of a default list of short instructions that contain no semantic restrictions and that together compute the desired expression. For example, consider a memory add-to-add instruction

madd k.2,k.1

This instruction can be encoded as the translation rules

$$\lambda := (:= \text{k.1} + \uparrow \text{k.1} \uparrow \text{k.2}), \text{"madd k.1,k.2"}$$

$$\lambda := (:= \text{k.1} + \uparrow \text{k.2} \uparrow \text{k.1}), \text{"madd k.2,k.1"}$$

The basic instruction pattern is ":= k + ↑ k ↑ k". If this pattern occurs in the IR, but the constants associated with each of the three k's are distinct, then the instruction cannot be used. The cure to this problem is to introduce new instructions with the patterns

$$\begin{aligned} r &::= \uparrow k \\ r &::= + \uparrow k\ r \\ \lambda &::= := k\ r \end{aligned}$$

that could be issued instead, simulating a non-restricted memory-to-memory add. The code generation could then proceed with these as though the longer instruction had been issued.

Glanville's table constructor constructs default instruction lists for all reduce actions having no semantically unrestricted instructions. The lists are obtained by simulating the action of the coder using as input the right hand side of the semantically restricted instruction and using onlv those instructions that are shorter than the one under consideration.

In the presence of semantically restricted instruction patterns (of which the table constructor has no knowledge), the code generator will choose one of a list of instructions in its reduce action. Assuming that this list has already been sorted by the table constructor, the instructions will be tested sequentially until one is found that is semantically compatible. In the event that no instruction is acceptable, a default list of substitutable instructions must be provided to implement that computation.

There are several classes of semantic restrictions that may have to be satisfied. Constants in the IR input may have to be equal to specific values, such as a 1 in an increment instruction, and logical registers may have to be equivalent to specific, actual registers. Multiple occurrences of a symbol in any class may have to refer to the same value or register. If the result is to appear in a register, then there must not be any references to the value in that register outside of that instruction pattern. Such references could exist only if some sort of common subexpression elimination has been done (see chapter 11). Finally, any additional semantic restrictions required to properly describe a particular target computer may be added to the code generator. The proof of correctness only requires that a default instruction or list of instructions be available for each semantically restricted instruction, so that it will always be possible to generate code.

Glanville shows that if looping and semantic blocking are eliminated by the extensions described above, and if the instruction description accurately describes the target machine, then the code generator produces correct code for all well-formed input.

The input IR is *well-formed* if

1. The input is a sequence of prefix expressions.
2. The operators and operands of the IR are from the same set as the operators and operands of the instructions and have the same meaning.
3. The sequence of prefix expressions is *valid*; that is, it is in the language generated or described by the instruction set.

Condition (3) must, of course, be checked by the implementor. The implementor can either prove that the routines generating the IR meet specification 1 or provide a simple routine to test the input to the code generator. It turns out that if the instruction set is uniform, condition 3 is also relatively easy to check.

10.8.2. Experimental Results

Several target computer descriptions, including the IBM 370 and the PDP-11 were input to a table constructor implemented in Pascal. Pascal

programs were manually translated into IR and input to the code generator. The PDP-11 code quality is competitive with the compiler "C" (Kernighan [1978]), yet uses no prior or post optimizations at all. Since the code generator produces assembly code, any postoptimizer can be used to further improve the code. On the PDP-11, some improvement can be obtained by exploiting short jump instructions and keeping track of register contents, for example. Similar results were found for the 370 code generator.

10.9. Bibliographical Notes

A good review of the uses of quads and triples in compiler construction and code improvement is given in Gries' text, "Compiler Construction for Digital Computers", Gries [1971], especially chapters 11, 13, 17, and 18. General reviews of code generation and the relation of machines to languages are given in four articles by Waite [1974a,b,c,d,e]. Poole [1974] discusses some compiler portability issues, including the use of intermediate languages and standard abstract machine descriptions.

Most of the material on the HP-3000, the CDC 6000 and the IBM System/360 is derived from manufacturer's manuals. The following is a minimum list:

- HP-3000 Computer System Reference Manual, part no. 03000-90019, Hewlett-Packard Co., 1501 Page Mill Rd., Palo Alto, CA., 94304.
- Control Data 6400/6500/6600 Computer Systems Reference Manual, Pub. No. 60100000, Control Data Corporation, 8100 34th Ave. So., Minneapolis, Minn. 55420.
- IBM System/360 Principles of Operation, Form A22-6821-3, IBM Corporation, Customer Manuals, Dept. B98, PO Box 390, Poughkeepsie, N.Y., 12602.
- IBM System/360 Operating System Assembler Language, Form C28-6514-4.
- IBM System/360 Supervisor and Data Management Services, Form C28-6646-2.
- IBM System/360 Operating System FORTRAN IV(G) Compiler Program Logic Manual, Form GY28-6638-1.

In addition, any technical library will contain textbooks on System/360 principles and programming. A good introductory treatment of the 360 architecture, assembler, loader conventions and macro processor is Donovan's text "Systems Programming", Donovan [1972].

Wirth's implementation of Pascal on the CDC 6400 is in his paper, "The Design of a Pascal Compiler," Wirth [1971a]. Glanville and Graham's work on a generalized code generator is in Glanville [1977].

An Algol 60 implementation on the IBM 360 is described in Bauer [1968].

CHAPTER 11

OPTIMIZATION

The term *optimization* in a compiler is applied to any special algorithm designed to yield more efficient object code than would be obtained by a simple, straight-forward code generator. An optimization may replace a simple coding algorithm by a more sophisticated one, or it may be an operation on some compiler data structure that transforms the structure into an equivalent but more efficient one.

The goals of optimization are the reduction of execution time and the reduction of code and data space. These two objectives are often found (unfortunately) to conflict. For example, execution time can often be reduced to some extent by adding more instructions. Thus a procedure call might be replaced by its code, thereby saving the execution time required to call a procedure. Ideally, a compiler user should be able to specify whether he wishes a minimum number of instructions or a minimum execution time, however this is seldom possible. Usually, some compromise is possible, and often a reduction in amount of code also results in a reduction of execution time as well.

Optimization is usually desirable because a simple code generator may fail to fully exploit the algebraic properties of the source language, or the full potential of the target machine's instruction set. For example, a direct translation of a statement like

```
A:=1+B+6;
```

would emit two additions. An optimization that exploits associativity of addition would reduce the translation to one addition. Also, a statement of the form

```
I:=I+1;
```

is very common in source programs. A general translation on a stack machine would yield the code

```
LOAD  I;
LOAD  =1;
ADD;
STOR  I;
```

Now many machines have "increment memory" instructions, so that the statement I:=I+1 could be translated into

```
INCM  I;
```

instead.

An optimization may be classified as *machine-dependent* or *machine-independent.*

Machine-dependent optimizations stem from special machine properties that can be exploited to reduce the amount of code or execution time.

Machine-independent optimizations depend only on arithmetic properties of the operations in the language and not on peculiarities of the target machine. Some common machine-independent optimizations are

- Constant arithmetic subsumption or folding, i.e., operations on constant expressions performed by the compiler rather than emitted as code.
- Identification and removal of identity or null operations. e.g., adding zero, multiplying by 1.
- Rearrangement of expression trees, exploiting commutativity, associativity or distributivity of certain operators, with the objective of reducing the number of operations.
- Identification of common subexpressions, for possible one-time evaluation.
- Elimination of useless or unreachable code.
- Movement of code from inside a loop to outside the loop, possibly changing its form to preserve the program semantics.

These are listed roughly in order of increasing complexity of their algorithms. Thus (1) and (2) are relatively easy to implement on an AST, (3) and (4) are somewhat harder, and (5) requires a control flow analysis of the loop statements.

We shall not attempt to develop or even classify all the possible optimizations. Nor will we present a systematic approach to optimization. We will discuss some of the most commonly used optimizations in sufficient detail for them to be useful in most compilers. Beyond that, the reader must refer to the voluminous literature on the subject.

11.1. Machine-dependent Optimizations

The nature of a machine-dependent optimization depends heavily on the target machine and the source language. A machine-dependent optimization may be applied to the AST by identifying special subtrees that happen to fit a machine feature, or it may be applied to an object code sequence. Still other optimizations deal with register and memory allocation and arrange that data appear in the appropriate registers in a reasonably optimal manner.

Some examples of machine-dependent optimizations are

1. *Register allocation.* In a typical computer, the most efficiently accessed memory is the most scarce resource (e.g., one or a few arithmetic registers, or the top-of-stack registers). The least efficiently accessed memory is the most plentiful (e.g., solid state memory, disk, drum, or tape). Most operations can be performed only on data in registers, and the consequent movement of data between registers and mass memory is a bookkeeping maneuver that contributes nothing directly to the computation. Hence the allocation of the scarce resources to achieve high efficiency in program execution is an important optimization.

2. *Special machine features.* Some common machine features (or *idioms,* a term first applied by Hall [1974]) that can potentially be exploited include: immediate instructions (a value is part of the instruction), incrementation (a memory location can be incremented by some constant), use of indexing or indirection, vector operations, etc.

3. *Data intermixed with instructions.* On many machines, data can be more efficiently accessed if it is intermixed with the instruction sequence. Some optimal arrangement of instructions and data therefore exists.

11.2. Machine-independent Optimizations

These optimizations are based on the mathematical properties of a sequence of source statements. An optimization essentially amounts to an analysis of the overall purpose of the statement sequence, then finding an equivalent sequence that will translate to the least amount of code.

Thus constant arithmetic can always be done by the compiler, rather than by emitting instructions to perform the arithmetic during execution. Arithmetic properties of the operations may also be exploited in the search for minimal code; their use might uncover some additional constant arithmetic, or some common subexpressions.

Whether an arithmetic property may be exploited depends on the specification of the language. If the language quite clearly specifies that all operations are to be performed from left to right in exactly the sequence dictated by the form of the expression, then arithmetic optimization is clearly limited. On the other hand, many modern languages specify that the operations may be performed in any order consistent with the arithmetic meaning of the expression.

Of the several arithmetic properties of the real numbers, the identity and null properties can always be exploited to reduce emitted code. Commutativity and associativity are very useful in subsuming constants, in identifying common subexpressions, and in optimizing register allocation. Distributivity is potentially useful in constant subsumption and subexpression recognition, but is difficult to exploit in a general way. The language may prohibit the use of distributivity for optimization—it may require that a parenthesized expression be performed first and not under any circumstances distributed among outer operations.

Computer arithmetic technically fails to satisfy any of the three properties of commutativity, associativity and distributivity, as the following three examples indicate:

Failure of Commutativity

Consider the statement

$$I:=I+F(A);$$

where F(A) is a function call that happens to change the value of I as a side effect. (I must be in a domain that is accessible to the function F.) On a stack machine, in the order shown, the emitted code might be

```
LOAD I;    {current value}
LOAD A;    {the parameter for the function call}
CALL F;
ADD;       {the change in I does not affect the copy
             previously written on the stack}
STOR I;
```

In commuted order, we would have instead:

```
LOAD A;
CALL F;
LOAD I;    {the new value}
ADD;
STOR I;    {a different result}
```

This side effect of the function call stems from an undesirable property of functions in several common languages, their ability to access and modify variables that appear with them in arithmetic expressions. For this reason, certain modern languages restrict functions to access only those variables local to the function call. Then this side effect could not exist; the compiler would be able to determine from the replacement statement alone whether the function could affect any other variables.

Failure of Associativity

Consider the following statements implemented on a machine with integer arithmetic in the range −32768 through 32767 (16-bit twos complement integers):

```
I:=30000;
J:=20000;
K:=21000;
I:=I-J+K;
```

If the last statement is executed in left-to-right order, the result 31000 is obtained without overflow. However, if the statement were instead

```
I:=I+K-J;
```

then the addition would overflow before the subtraction is performed.

Here, if the language specification permits associativity for the sake of optimization, the programmer is at fault in not considering the possibility of overflow.

Failure of Distributivity

Consider the following example:

```
I:=15;
J:=4000;
K:=3000;
I:=I*(J-K);
```

The last statement, if performed in the order indicated by the parenthesizing, will yield 15000 without overflow. However, the distributive equivalent

```
I:=I*J - I*K;
```

will overflow twice.

Other sources of failure of computer arithmetic to satisfy mathematical properties stem from (1) integer division, where the fractional part is truncated to yield the largest integer less than the quotient, and (2) floating-point subtraction, where the two numbers are very nearly alike; the result will likely have lost all significance.

Despite these difficulties, algebraic optimization is often provided by a compiler for those languages that permit it. Where optimization is provided, the programmer must be alert to the potential failure of his coded algorithm; the optimizing compiler may generate a somewhat different algorithm than he coded.

11.2.1. Expression AST Optimizations

Some arithmetic optimizations are most effectively performed on an abstract-syntax tree for an arithmetic expression. Let us consider some of these, for a set of operations that includes the four binary arithmetic operations and unary minus. A similar principle applies to binary and unary logical operations.

The easiest optimizations are those based on the identity and null properties of 0 and 1. We need only inspect the environment of a constant 0 or 1 in the AST and apply one of the operations indicated in figure 11.1. Here, T stands for some subtree of arithmetic expressions. These obviously express the identities

$$
\begin{aligned}
&0+T = T+0 = T\\
&T-0 = T\\
&0-T = -T\\
&0*T = T*0 = 0\\
&0/T = 0 \quad \{\text{however, } 0/0 \text{ is undefined}\}\\
&T/0 = \infty \quad (\text{or overflow})\\
&1*T = T*1 = T/1 = T
\end{aligned}
$$

Such expressions are not likely to be written directly by a knowledgeable programmer, but frequently arise through the use of macros with parameters. Without compiler optimization, the macro definition must contain many special tests in order to yield reasonably optimal code.

Constant evaluation is facilitated by a rearrangement of the AST designed to bring constants together. For example, the expression

$$4*5+I-6$$

has the AST shown in figure 11.2. In this form, only the $4*5$ evaluation is possible. However, associativity may be applied to the $+$ and $-$ to yield the AST shown in figure 11.3; the multiplication and subtraction may then be

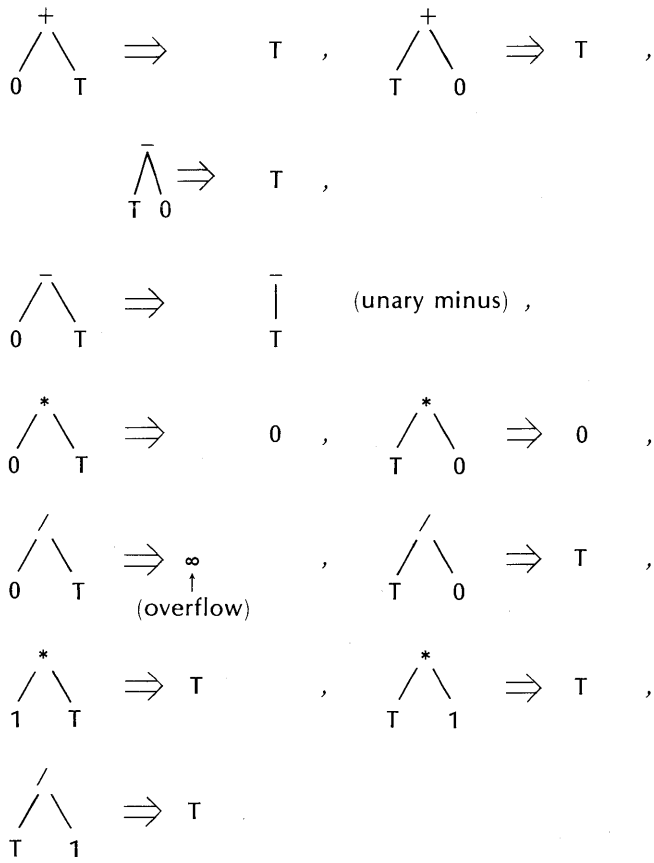


Figure 11.1. Abstract-syntax tree (AST) optimizations based on arithmetic properties of 0 and 1.

performed to yield the the tree for 14+I. A general application of commutativity and associativity in an AST is called *flattening*.

11.2.2. Flattening

In general, a set of connected + and − nodes may be collected into a single node with more than two children; the result is to *flatten* an AST. The flattened tree then facilitates the recognition of optimizations through associativity and commutativity.

An *addition cluster* of nodes in an AST is a set of nodes A such that (1) every member of A is in {+, −}, and (2) the set A forms a tree. Then a cluster A may be reduced to a single node by applying the transformations shown in figure 11.4. For example, a binary − node can be replaced by a

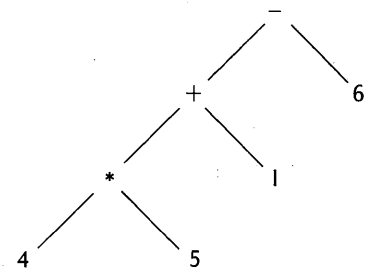


Figure 11.2. AST for an expression 4*5+1−6.

binary + node by negating the second child of the − node. The overall effect of the transformations of figure 11.4 is to obtain as large a cluster of + nodes near the top of a tree as possible, with unary − nodes pushed down as much as possible.

A cluster of * and / nodes can be formed through similar transformations. A binary division A/B can be replaced by A*(1/B), provided that the division is performed in floating-point and not fixed-point.

Once a cluster of additions or multiplications is identified, the AST may be flattened by merging each cluster into a single node, with several children. Thus the expression

$$A*B*C/D + E*F - D*A$$

has the flattened tree shown in figure 11.5. The multiple children of some + or * node may then be rearranged in any manner to achieve an optimization. For example, the constant children can be placed to the right (or left) of variable children and subsumed into a single constant node.

Parenthesized expressions must be protected (in most languages) from cluster merging. Thus the tree for A+(B+C) cannot be flattened into a

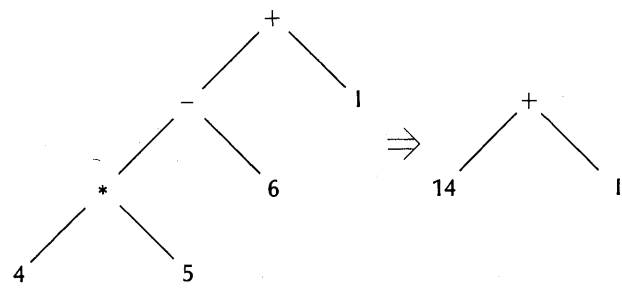


Figure 11.3. Associativity of {+,−} applied to AST of figure 11.2.

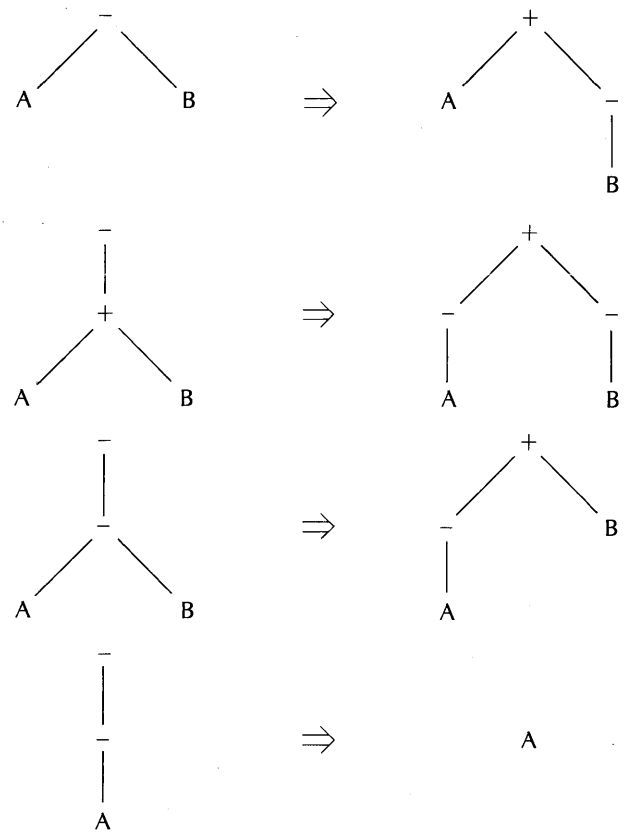


Figure 11.4. Transformations applied prior to flattening an AST for binary operations $\{+,-\}$.

single + node. A node representing a parenthesized expression must therefore carry a mark that prevents its merger into a cluster.

Flattening is also useful for single-register machine coding. In such a machine, a binary arithmetic operation applies to an accumulator register A and some memory location M; the result is left in A. Initially, A must be loaded from memory. Some expressions require temporary memory cells T_1, T_2, ... Thus the replacement statement

$$W:=R+S+U*V;$$

would be coded without optimization as:

```
LOAD  R;   {value of R to accumulator}
ADD   S;   {add S to accumulator}
STOR  T1;  {R+S must be saved to do
             the multiplication}
```

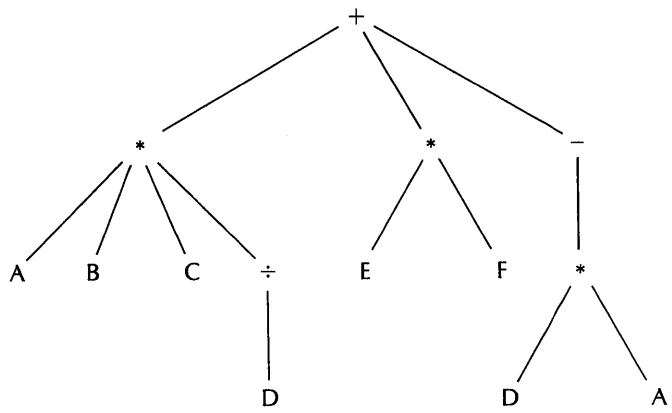


Figure 11.5. A flattened AST for the expression A*B*C/D + E*F − D*A.

```
LOAD  U;
MPY   V;
STOR  T2;
LOAD  T1;
ADD   T2;
STOR  W;  {9 instructions, 2 temporaries needed}
```

The flattened AST for this expression is shown in figure 11.6. For optimal single-register code, the * subtree should be the leftmost sibling of the + node, as indicated in figure 11.7. Then the single-register code for the tree of figure 11.7 is:

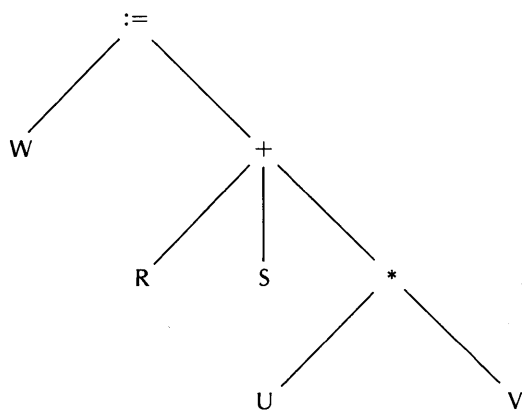


Figure 11.6. Flattened AST for statement W:=R+S+U*V.

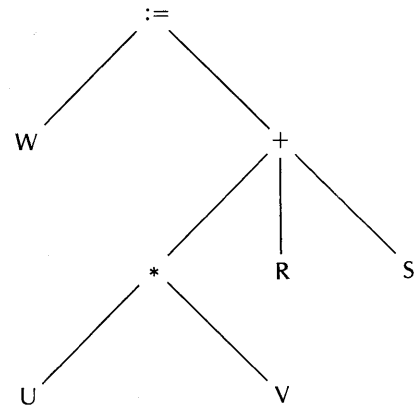


Figure 11.7. AST of figure 11.6 rearranged for optimal single-register code.

```
LOAD  U;
MPY   V;
ADD   R;
ADD   S;
STOR  W;  {5 instructions, no temporaries}
```

A reasonable optimization rule is therefore: (1) transform binary subtraction and binary floating-point division nodes into binary addition and multiplication nodes, in order to produce as many clusters as possible, (2) merge the clusters, (3) arrange constant siblings of some cluster node to the right and complicated trees to the left. (We shall develop a more complete optimal algorithm later.)

Exercises

1. Design an algorithm that identifies clusters in an arithmetic AST and flattens them. Assume floating-point arithmetic only.
2. Discuss the possibilities of optimization of each of the following operations:
   (a) A ↑ B {exponentiation}
   (b) if L then A else B {selection of A or B, depending on L}
   (c) case I of N1: A; N2: B; N3: C; . . . Ni: Z; end; {multi-way selection of an expression}

3. Design an algorithm that identifies and reduces an AST containing identity and null operations, e.g.,

$$0*X,\ 0+X,\ 1*X,\ X/1,\ X-X,\ \text{etc.}$$

4. Show that commutativity and associativity can yield no improvement in the number of instructions for a single expression tree, on a pure stack machine with an unlimited number of registers. A *pure* stack machine has an unlimited stack and the instruction set:

LOAD n	{C(n) → C(tos); tos = address of stack top}
STOR n	{C(tos) → C(n)}
ADD	{C(tos − 1) + C(tos) → C(tos − 1); tos: = tos − 1}
SUB	{C(tos − 1) − C(tos) → C(tos − 1); tos: = tos − 1}
MPY	{C(tos − 1) * C(tos) → C(tos − 1); tos: = tos − 1}
DIV	{C(tos − 1) / C(tos) → C(tos − 1); tos: = tos − 1}

5. Suppose that, in addition to the instructions given in exercise 4, a stack machine has some memory reference instructions that operate between a memory location and the stack top. Would it be possible to optimize evaluation of an expression tree using these memory reference instructions? If so, give some examples. The additional instructions would have the form:

ADDM n	{tos + C(n) → tos}
SUBM n	{tos − C(n) → tos}
MPYM n	{tos * C(n) → tos}
DIVM n	{tos / C(n) → tos}

11.3. Optimal AST Evaluation for a Multiregister Machine

The reported work in code optimization can be divided into two general classes: the class of *code-improvement methods* and the class of *provably optimal methods*. The latter class is much smaller than the former, undoubtedly because it is relatively easy to develop a heuristic method that, when it applies, improves the efficiency of the emitted code; it is much harder to prove that some general algorithm yields a code sequence that is absolutely minimal. Indeed, several important optimization questions are either undecidable or can only be solved by an algorithm of exponential complexity, a so-called NP-complete algorithm [Aho 1974b]. In the face of undecidability

or an NP-complete problem, heuristic code-improvement represents the only practical approach to an optimization.

One of the few optimal code generation algorithms is from Meyers [1965] and Nakata [1967], as expressed and analyzed by Sethi and Ullman (Sethi [1970]). This algorithm generates optimal code from a binary expression tree, for a multiregister machine. Three variations on the basic algorithm are developed, one that assumes that no operation is commutative or associative, a second that assumes that some operations are commutative, and a third that assumes that some are both commutative and associative.

It is important to note that Sethi's work does not address the problem of expression tree optimization, where the tree contains common subexpressions or of a set of expression trees that might be partially merged. These questions will be examined later.

11.3.1. The Machine

The commands permitted by the Meyers-Nakata-Sethi algorithm are of four types, for a machine with an unlimited amount of general memory and NR $\geq$ 1 general purpose registers:

LOAD M,R: Copy contents of memory cell M to register R.

STOR M,R: Copy contents of register R to memory cell M.

OP R1,M,R2: Perform a binary operation OP (non-commutative in general) on register R1 and memory M, the result to replace contents of register R2. Neither R1 nor M is changed by the operation, unless R1 = R2.

OP R1,R2,R3: Perform a binary operation on registers R1 and R2, the result to replace contents of register R3. Any two or three of the registers may be the same. The contents of R1 and R2 are unchanged, unless one or both is register R3.

Note that a binary operation OP R1,M,R2 does not have a counterpart OP M,R1,R2. For example, the machine may have a subtraction operation R − M, but no "reverse" subtraction M − R. This restriction exists in many commercially available computers, and is therefore a realistic one to consider. If both operations exist, then every operator could be treated as though it were commutative.

Let us assume that the tree's value will be left in a register. Then some trees will require no STORs; there will be sufficient registers to hold all intermediate values. However, no matter how large NR is, there are trees that will require some STORs of intermediate results. The problem is to identify those leaf nodes that must be LOADed, those that must be stored, and the order of performing all the operations. Every operator node in the tree will

require one OP instruction regardless of the manner of loading operands and storing results, so that a minimal solution is one with the fewest LOAD and STOR instructions.

Considering the number of degrees of freedom possible, it is remarkable that an algorithm that finds the minimal code sequence in a reasonable time exists. We shall see later that if common subexpression optimization is also desired, then no algorithm that finds a minimal code sequence in a reasonable time exists—any such algorithm will require an exponential time complexity to find an optimal solution.

11.3.2. Tree Labeling

We begin with an algorithm that decorates each node N of the expression tree with the integer quantity LABEL(N). This quantity will represent the number of registers needed to evaluate the subtree rooted in N without any STOR instructions. The algorithm invokes several tree functions explained in the comments.

```
procedure TREELABEL(N: integer): integer;
  {a recursive procedure that accepts a
   node N and returns a label for the node}
begin
  if LEAF(N) then
      {LEAF(N) is true if N has no
        children}
     if N ≠ ROOT then  {ROOT = AST tree root}
        return(if N=LEFTCHILD(PARENT(N)) then 1 else 0)
             {PARENT(N) is the parent of node N;
              LEFTCHILD(N) is the left child of N}
     else return(1)  {tree consists of one node}
  else
     begin  {N has children}
        var L1, L2: integer;  {two temporary labels}
        L1:=TREELABEL(LEFTCHILD(N));
        L2:=TREELABEL(RIGHTCHILD(N));
           {get the labels of the child subtrees}
        return(if L1=L2 then L1+1 else
               MAX(L1,L2))  {maximum of the two labels}
     end
end   {of procedure}
```

TREELABEL is a bottom-up algorithm; the label for some node can be determined only if it is a leaf node or if the labels for its children are known.

Suppose that a node N is a leaf node. If a left leaf, then we need one register to hold its value in order to perform its parent operation. If a right leaf, its parent operation can be performed from memory, requiring no register. Of course, the left subtree of an operation with a right child leaf will require at least one register.

Now suppose that node N is an internal node. Let its child subtrees carry labels L1 (left) and L2 (right). If $L1 = L2$, then an equal number of registers are required to evaluate each of the subtrees; note that in this case L1 and L2 must be at least 1. We need one more register to carry out the operation of node N, to hold the value of the left subtree while the right subtree is being evaluated, or vice versa, hence the label for N is $L1 + 1$. If $L1 \neq L2$, then at least one register is available for one of the subtrees (the one with the lesser label); no additional registers are needed to evaluate node N, hence $L(N) =$ MAX(L1,L2).

11.3.3. Optimal code generation

We now introduce an optimal code generation algorithm, procedure EVALUATE. This is a top-down algorithm that operates on an expression tree decorated with the labels developed in the TREELABEL procedure above. It requires some more functions as follows:

EMITLOAD(M,R): Emits a LOAD instruction, LOAD M,R.

EMITSTOR(M,R): Emits a STOR instruction, STOR M,R.

EMITOP(OP,R1,S,R2): Emits a binary operation OP R1,S,R2; OP is the operation, R1 and R2 are registers, and S is a or a memory location.

OP(N): This is the operator associated with node N. (N must be an internal node.)

LABEL(N): The label associated with node N, an integer, as determined by TREELABEL.

MEMLOC(N): The memory location associated with node N; N must be a leaf node. We assume that every leaf node is associated with a memory location; no variable is already in a register.

ALLOCATE: A typed procedure that returns a temporary memory location. Temporary locations are needed when the register set cannot hold all the intermediate tree values.

RELEASE(T): Returns a temporary location T to a pool. Those in the pool may subsequently be used for other intermediate results.

Now the code emitting algorithm is as follows. The comments {A}, {B} indicate sections of the algorithm that will be expanded upon in the next section.

```
   procedure EVALUATE(N, M: integer);
      {N is a subtree root node.
       M is the first of a set of registers
         available for the evaluation of the
         subtree rooted in N; the available
         registers are M, M+1, . . . ,
         where the machine contains NR
         registers.  Initially, EVALUATE is
         called with N=ROOT and M=1.  The
         result of an EVALUATE call is the
         value of the subtree N left in
         register N}
   begin
      if LABEL(N)=1 then
      begin
         if LEAF(N) then
         begin  {N must be a left child}
{A}      EMITLOAD(MEMLOC(N), M)
         {a left leaf must be LOADed}
         end
        else
         begin {the right child must be a leaf}
{B}         EVALUATE(LEFTCHILD(N), M);
            EMITOP(OP(N), M,
                 MEMLOC(RIGHTCHILD(N)), M)
         end
      end
     else
      begin  {LABEL(N) > 1 here;
              N must have children}
         var N1, N2, L1, L2: integer;

         N1:=LEFTCHILD(N);
         N2:=RIGHTCHILD(N);
         L1:=LABEL(N1);
         L2:=LABEL(N2);

         if MIN(L1,L2) ≥ NR then
         begin
            var T: integer;
```

```
{C}         EVALUATE(N2, M); {evaluate right child}
            T:=ALLOCATE; {allocate a cell}
            EMITSTOR(T,M); {register M put away}
            EVALUATE(N1,M);
            EMITOP(OP(N),M,T,M);
            RELEASE(T) {T no longer needed}
          end
        else
            if L1 ≠ L2 then
{D}         begin  {Note: MIN(L1,L2) < NR here}
            if L1 > L2 then
                N1 :=: N2; {exchange N1, N2; now
                        N1 is the node with the least label}
            if MIN(L1,L2) > 0 then
            begin
                EVALUATE(N2,M);
                EVALUATE(N1,M+1);
                if L1<L2 then
                EMITOP(OP(N),M+1,M,M)
              else EMITOP(OP(N),M,M+1,M);
            end
           else
            begin {lesser of L1, L2 is 0}
                EVALUATE(LEFTCHILD(N),M);
                EMITOP(OP(N), M, RIGHTCHILD(N),M);
                    {N's right child must be a leaf}
            end
          end
        else
{E}         begin  {L1 = L2 and L1 < NR}
            EVALUATE(N1, M);
            EVALUATE(N2, M+1);
            EMITOP(OP(N),M,M+1,M)
          end
      end  {of procedure}
```

11.3.4. Discussion

Section {A} deals with the case of a leaf node N with label 1. N must be a left child of its parent. No operation has its left operand in memory, so we must load the value of N. We use the next available register M for this purpose.

In section {B}, the right child of N must be a leaf; the label of N cannot be 1 otherwise. We can evaluate the left child of N with registers M, M+1, . . . , NR available, and the operation of node N can then be done with no additional registers.

In section {C}, both children of N require more registers than are available. We must therefore allocate a temporary cell T and use it to save register M. We then evaluate the right subtree and save the result. When the left subtree is evaluated, its result will be in register M, ready for the operation OP(N). After the operation, the temporary cell T is no longer required. Note that in the Pascal procedure, T is a local variable and is therefore preserved in the recursive calls.

In section {D}, one of the labels (L1, L2) is less than NR. This means that at least one register is available for the operation OP(N). A special case arises if the lesser label is zero; it must be the label of the right child, a leaf node. We can therefore evaluate the left subtree with registers M, M+1, . . . available and then operate directly on the register M and the right child leaf.

If the lesser label is not zero in section {D}, we evaluate that subtree with the lesser label first, then the subtree with the greater label. The former result is left in M and the latter in M+1. Finally, the operation OP(N) is a register-register operation, with the result in register M.

In section {E}, the subtree labels are equal and less than NR. We therefore have an available register for the operation, which will be a register-register operation.

Example. Consider the expression tree of figure 11.8. The nodes are named

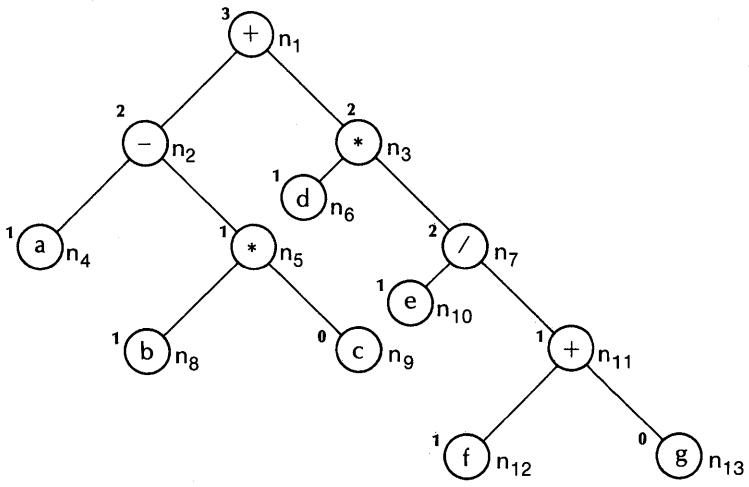


Figure 11.8. An example expression tree labelled with the minimum number of registers required to evaluate each subtree.

$n_1, n_2, \ldots n_{13}$, and carry labels ranging from 0 through 3. The leaf nodes carry operands a, b, c, . . . , g. The tree represents the expression (a−b*c)+(d*e/(f+g)).

For example, the root node carries label 3 since its children each carry label 2. Node n_3 carries label 2 since its children carry labels 1 and 2 respectively.

We may trace the recursive calls of EVALUATE as follows, for a machine with NR=2 registers, R1 and R2:

EVALUATE(n_1, 1) yields the calls and code:

```
EVALUATE(n3, 1);
STOR T1,R1;
EVALUATE(n2, 1);
ADD  R1,T1,R1;
```

EVALUATE(n_3, 1) yields the calls and code:

```
EVALUATE(n7, 1);
EVALUATE(n6, 2);
MPY  R2,R1,R1;
```

etc. (See Exercise 1)

When all the EVALUATE calls are traced, the following code sequence is found:

```
LOAD  e,R1;     {load operand e}
LOAD  f,R2;
ADD   R2,g,R2;     {e.g., C(R2) + g → C(R2)}
DIV   R1,R2,R1;
LOAD  d,R2;
MPY   R2,R1,R1;
STOR  T1,R1;
LOAD  a,R1;
LOAD  b,R2;
MPY   R2,c,R2;
SUB   R1,R2,R1;
ADD   R1,T1,R1;  {result left in register R1}
```

Since the tree carries a maximum label 3, at least one temporary location is needed in any code sequence.

Exercises

1. Complete the list of EVALUATE calls and verify the code sequence given above.

2. Develop optimal code sequences for machines with NR = 1 and NR = 3, for the AST of figure 11.8.
3. Prove that EVALUATE evaluates a tree with no stores when as many registers as the label of the root node are available. (*Hint:* Is the label of the root node the largest of any of the tree labels?)
4. A *minor node* is a leaf node that is the left child of its parent. Show that the number of LOAD's in an evaluation program must be at least equal to the number of minor nodes in the expression tree.
5. A *major node* is an internal node, both of whose children have LABEL values greater than or equal to NR, the number of registers in the machine. Show that the number of STORs in an evaluation program is at least equal to the number of major nodes in the tree.
6. Prove that EVALUATE generates an optimal code sequence. *Hint:* Use examples 4 and 5 above, and prove the following two lemmas:

   Lemma 1. The number of stores generated by EVALUATE is equal to the number of major nodes in the tree.

   Lemma 2. The number of loads generated by EVALUATE is equal to the number of minor nodes in the tree.

11.3.5. Commutative Operators

When a commutative operator is present in an expression tree, we have an opportunity to interchange its children with the objective of reducing the number of loads or stores. Considering Lemmas 1 and 2 in exercise 6 above, we see that the number of stores is unaffected by such an interchange. However, the number of loads can be reduced, if the number of minor nodes is thereby reduced. Clearly, those commutative operators that carry a leaf left child and a nonleaf right child are candidates for child interchange, and the interchange will reduce the number of loads. We therefore have the rule:

Commutative Operator Reduction Rule. For every node N such that: (1) OP(N) is commutative, and (2) LEAF(LEFTCHILD(N)), and (3) $\sim$LEAF(RIGHTCHILD(N)), do: interchange (LEFTCHILD(N), RIGHTCHILD(N)).

The resulting tree will yield an optimal code sequence under algorithm EVALUATE.

For example, the tree of figure 11.8 is transformed into the tree of figure 11.9 through an interchange of the subtrees rooted in nodes n_6 and n_7. We thereby reduce the code sequence by one LOAD.

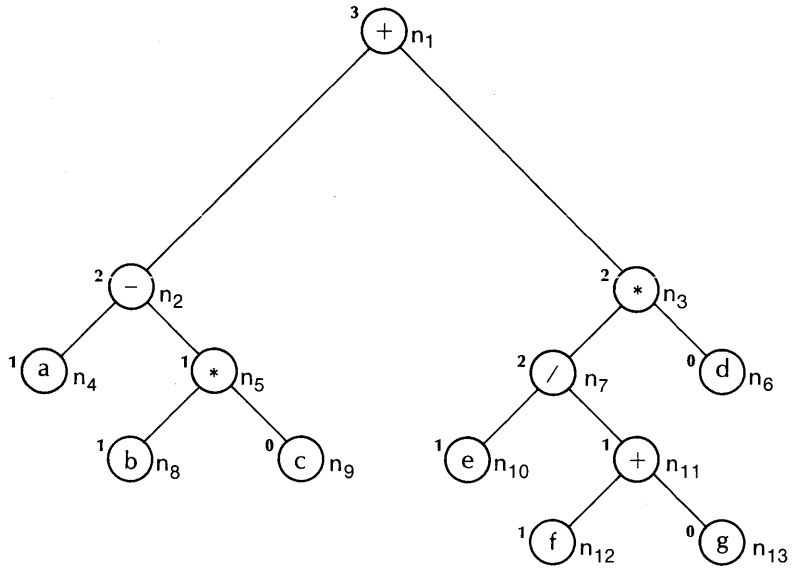


Figure 11.9. Commutative optimization of tree of figure 11.8.

11.3.6. Associative and Commutative Operators

Associativity by itself is of little value in reducing the number of loads and stores in a code sequence; however, the combination of associativity and commutativity (A–C) is of value.

We use the notion of a cluster developed earlier. A *cluster* is a set of A–C nodes associated with the same operator that forms a maximal tree. When some clusters exist, the tree may be reorganized in any fashion by rearranging the subtrees of any cluster.

The labeling rules for an associative tree are slightly different than those expressed in TREELABEL. The children of every node are first arranged in decreasing label sequence from left to right (the left most child has the largest label). Let L_1 and L_2 be the labels of the two left most children of some node N. Then LABEL(N) $= L_1$ if $L_1 > L_2$, and $L_1 + 1$ if $L_1 = L_2$.

Given such a labeled associative tree and any cluster node N in such a tree, e.g., figure 11.10(a), we then transform the cluster subtree into the binary subtree shown in figure 11.10(b), with root N′. This transformation is made on every associative cluster node. The resulting tree then yields a minimal code sequence under the EVALUATE algorithm.

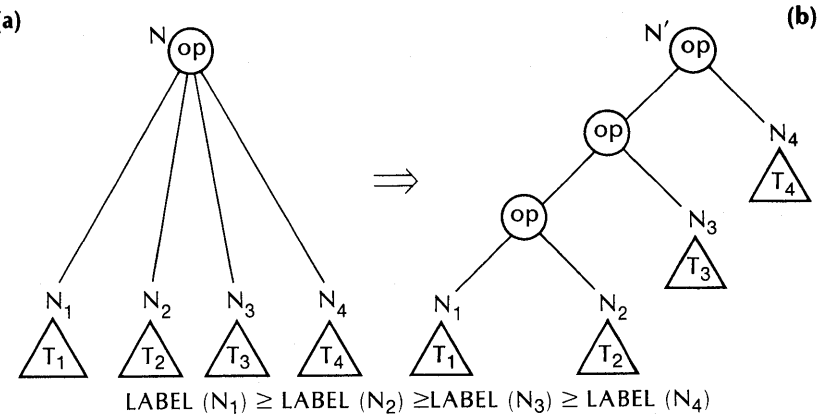


Figure 11.10. Transformation of an associative tree into an optimal binary tree.

Exercises

1. Construct a tree for the expression

$$a*(b+(c*d)+e)*f+g+h*j$$

   and apply the associative-commutative transformations described above. Generate the minimal code sequence for the expression.

2. Show that the associative-commutative transformation yields a minimum code sequence.

11.4. Code Improvement over a Sequence of Statements

The generation of optimal code for a single expression is hardly worth while in an average program. Knuth [1971], in a study of a large sample of FORTRAN programs, found that of all the assignment statements in his sample, 68% were a simple replacement of the form A = B, with no arithmetic operators and 13% were of the form A = A op B, with the first operand on the right the same as the replacement variable. Hence only 19% have a more complex structure, most of which apparently involve very few operators.

Of the variables, 58% were not indexed, 30.5% had a single index, 9.5% had 2 and only 1% had 3 indices (the maximum number under the Fortran in use at Stanford and Lockheed, the sources of the test programs).

An optimization for expression trees only will probably have little effect on the efficiency of a typical program. Usually few statements will have an

expression tree large enough for the algorithm of the preceding section to yield an improvement. Of course, these expressions might reside in a frequently executed section of the program, magnifying their importance to the program's performance.

The prospect of eliminating common subexpressions and register assignment over a sequence of assignment statements, called a *block*, is more promising.

Knuth also studied the improvement that would result from different levels of optimization. At the lowest level, blocks were considered for constant subsumption, rearrangement and redundancy elimination. A reasonable register allocation strategy was also employed. At a second level, the flow of control among blocks was also considered to achieve global constant subsumption and rearrangement and some strength and frequency reduction. Improvements were found at each level, the second yielding somewhat greater relative improvement (40% increase and 170% increase over the raw non-optimized code, respectively). A further improvement could be made by exploiting idioms for the IBM System/360 system, other than its basic multiregister organization.

Code can be improved in a sequence of statements in several ways: a larger field of expressions is available for identifying common subexpressions; register allocation (for a multiregister machine, at least) can be improved; and loop-invariant expressions can be moved outside a loop.

None of these is possible without the development of methods for determining the status of all the program variables at any one point. For example, the value of a variable may by undefined at certain locations in the program, then defined later. A given definition will hold through some sequence of statements, then be replaced by another definition. Finally, a variable may become *dead* after some reference, i.e., it is no longer used after the reference. The status of the variables obviously influences register allocation and the identification of common subexpressions and of loop invariants.

We see that development is needed along three lines. We need: (1) a more general optimization plan for a block, that can identify common subexpressions and allocate registers efficiently, (2) an analysis of program control flow and its effect on the status of each variable, and (3) the exploitation of variable status in intra-block as well as inter-block optimization.

11.4.1. Blocks

A block consists of a sequence of assignment statements $S_1, S_2, \ldots, S_b$ that are executed in that order, and such that control can only pass into the first statement, from any one statement to its successor, and out of the last statement. No program branch may enter the block except to its first statement.

Each statement is a simple assignment of the form

$$C \leftarrow \theta\ B_1\ B_2 \ldots B_n$$

where θ is an n-ary operator, $B_1, B_2, \ldots, B_n$ are its operands, and C receives the single scalar result of the operation. C is said to be *defined* by the statement, and the B_i are said to be *referenced* or *used*.

This schema is sufficiently general to cover many common program statements. A simple replacement statement of the form

$$X := A+B*C$$

can be subdivided into a sequence of simple assignments by introducing a temporary variable T:

$$T \leftarrow *, B, C$$

$$X \leftarrow +, A, T$$

A function call returns a single value; however, if some of its parameters are called by reference or by name, then the possibility exists that they have been defined. Such parameters may be marked *defined* by the null operation DEF:

$$B \leftarrow DEF$$

indicates that B has been defined in some previous statement, such as a function call:

$$R \leftarrow FN, P_1, P_2, \ldots B, \ldots, P_r$$

An indexed replacement, e.g.,

$$B(X) := E$$

involves an assignment for X and then for B; however, since the compiler cannot determine which of the vector of B variables has been defined, it can only assume that any one of them has been. An array variable is therefore referenced or defined as a unit; the optimization algorithms cannot usually distinguish the members of the vector.

A vector move statement is similarly treated. Thus in PL/I, if A and B are compatible structures, the statement

$$A = B$$

represents an assignment to A of each of the components of B, and the optimization algorithms must deal with structures A and B as units.

11.4.2. Variables and Their Domains

A *variable* is some value created at run-time by a definition that is

preserved until another definition or the end of the program is reached. A variable is associated with some name, but a given name may represent several different variables.

Thus in the following statements, one variable associated with X is marked by the arrows (↑):

```
X := X + Y;
↑
A := X + 5;
     ↑
X := X - Y;
     ↑
```

Note that the first reference and the last definition of X are not part of this variable; they are part of another variable, one that happens to carry the same name in these statements.

The *domain* of a variable is the set of all statements in the program that includes the definition statement, every reference statement, and every statement S such that a control path may pass from the definition through S to some statement containing a variable reference. The variable is said to be *live* at some statement if the statement is within its domain.

If the domain of a variable extends through the end of a block B, the variable is called an *output* variable of B. If the domain includes any statement passed through before reaching the beginning of the block, the variable is an *input* variable of B. All other variables are *local* or *temporarq* variables; their domains are contained within the block.

The input variables of a block can be identified—they have some reference prior to a definition. Unfortunately, the output and local variables cannot be distinguished from an examination of the block alone; it is necessary to examine the control paths that lead from the block. Certain local variables can be identified: (1) the declared local variables, for a source language that permits variables to be declared at the head of any block, and (2) temporaries introduced through the reduction of arithmetic expressions to simple assignment statements.

11.4.3. Equivalent and Normal Blocks

Two blocks are said to be *equivalent* if they carry the same sets of input and output variables, and if, for every set of values of input variables, the resulting output variable values are the same upon execution of the block statements. One of the blocks may always be transformed into the other through a sequence of four equivalence transformations, as Aho [1974b] has shown. The four transformations are

1. Removal of useless statements and variables.
2. Identification of two computations producing the same value (common subexpressions).
3. Renaming of variables, e.g., temporary variables.
4. Interchange of two adjacent statements, under conditions that ensure preservation of equivalence.

A *normal* block is such that every variable carries a unique name. Any block can be transformed into an equivalent normal block by renaming certain of the temporary or input variables. For example, consider the block

```
X:=A+B;
Y:=A-B;
X:=X*Y;    {second use of variable name X}
Y:=Y*X;    {second use of variable name Y}
```

The names X and Y are associated with more than one variable each in this block; we therefore introduce new names X′ and Y′, to yield the following equivalent normal block:

```
X:=A+B;
Y:=A-B;
X′:=X*Y;   {beginning of second variable X′}
Y′:=Y*X′;  {beginning of second variable Y′}
```

An input variable X must be renamed if a definition of X appears in the block. An input variable that is also an output variable need not be renamed. We adopt the convention that output variables will never be renamed.

Only normal blocks will be considered in the following sections.

11.4.4. Representation of a Block as a DAG

A normal block may be represented as a directed acyclic graph, or DAG, in much the same way that an expression can be represented as a tree. A DAG consists of a number of nodes connected by directed edges, such that no directed path is closed. A DAG will have one or more root nodes, with no indirected edges, and one or more leaves, with no outdirected edges. A node with at least one outdirected edge is an *internal* node. (The root nodes may be internal nodes). We will be interested only in DAGs for which every root is not a leaf.

Each node of a block DAG will be associated with a variable. In addition, each internal node will be associated with an operation.

A DAG is constructed from a normal block as follows:

1. The statements $S_1, S_2, \ldots, S_b$ of a block will be considered in that order. Each statement will in general add nodes and edges to the block DAG.
2. Given a statement

$$C \leftarrow \theta B_1 B_2 \ldots B_n$$

we add a new leaf node for every variable B_i not already in the DAG. A new node N_C for variable C and operation θ is added to the DAG. Then edges from node N_C to each node $B_1, B_2, \ldots, B_n$ are added.For example, consider the block

T ← +, A, B
C ← *, T, A
A ← /, C, A
F ← +, E, E
B ← /, T, D

where {A, B, D, E} are input variables and {A, B, F} are output variables. Since A and B are redefined in the block, we must rename the corresponding input variables. The equivalent normal block is

T ← +, A′, B′
C ← *, T, A′
A ← /, C, A′
F ← +, E, E
B ← /, T, D

This block has the DAG shown in figure 11.11.

11.4.5. Value of a DAG

The *value of a DAG* is a set of values associated with its nodes. These values are determined by the values associated with the input variables and a DAG evaluation rule. Each node of a DAG is associated with a value, computed by the following rule:

DAG Evaluation Rule. If the node is a leaf, it must be an input variable; its value is therefore the value of that variable. If the node is internal, its value is the result of its operator applied to its children's values.

An input node's value clearly cannot be determined until each of its

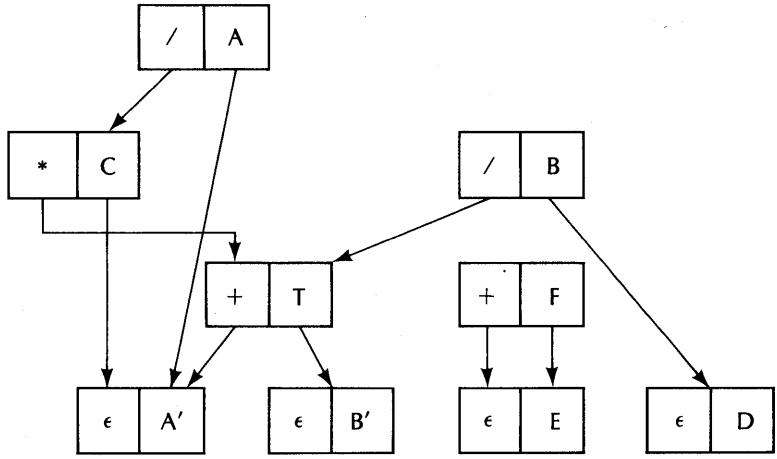


Figure 11.11. A directed acyclic graph (DAG) for the block {T ← A′+B′, C ← T*A′, A ← C/A′, F ← E+E, B ← T/D}.

children's values are determined. Now each node represents a unique variable because of block normality. Evaluation of some node results in fixing the value of its associated variable; however, no other node evaluation can affect that variable's value, hence every evaluation is uniquely determined, regardless of the order in which evaluation occurs. The DAG value is also clearly the value returned by the block.

We are therefore free to explore DAG evaluations without regard to the order in which the evaluations take place. Evaluation is subject only to the constraint that a node can be evaluated only if all its children have been previously evaluated.

Furthermore, all of the internal non-output nodes represent temporary variables. These may be assigned to registers and never be allocated a temporary memory location, for the sake of efficiency, even though the source program may have assigned a name to the variable.

11.4.6. Common Subexpression Identification

During construction of a DAG, the appearance of two subexpressions, such that the second one need not be recomputed, is easily detected by a minor change in the construction algorithm.

We first suppose that none of the block operators are commutative or associative. (This restriction will be lifted later.) We are interested in identifying a DAG node that represents a value that can be saved for use later in the block. Such a node represents a subexpression that appears twice in the block and that, in addition, will return the same value on its second appearance.

We are therefore interested in a pair of assignment statements

$$S_i: \quad C \leftarrow \theta B_1 B_2 \ldots B_n$$
$$S_k: \quad C' \leftarrow \theta B_1 B_2 \ldots B_n$$

such that none of the B_m ($1 \leq m \leq n$) are defined in any statement S_j where $i < j \leq k$. That is, statement S_k will return the same value to C′ as statement S_i returned to C only if none of the referenced variables have been redefined in the interim.

Now block normality guarantees that none of the B_i are redefined; if one were, we then have a reference of that variable prior to its definition and its name could not appear in both S_i and S_k. The mere existence of the pair of statements $\{S_i, S_k\}$ is sufficient to guarantee that C′ will receive the same value as C. In terms of the DAG being constructed, when statement S_k is under consideration, each of the nodes $B_1, B_2, \ldots, B_n$ will already be in the DAG, and a one-level tree with a root associated with operator θ and variable C will exist. We need only look for this situation, and add variable C′ to the node that contains C.

We see that a node will carry more than one variable in this extension in general.

11.4.7. Use of Associativity and Commutativity

Suppose that some of the operators are commutative and still others are associative-commutative. We may identify common subexpressions that are equivalent under these arithmetically invariant transformations by two simple operations.

Before the DAG is built, the operands of every commutative operator are arranged in some canonical order. For example, they may be arranged in alphabetical order. Then when the DAG is built, a pair of commutatively equivalent nodes will be identified as common subexpressions.

After the DAG is constructed, we search for associative clusters of nodes. Two nodes P and Q belong to the same associative cluster if and only if: (1) an edge from P to Q exists, (2) they carry the same associative operator, (3) excluding P, Q has no ancestors, and (4) at most one of them carries an output variable. If P and Q belong to the same cluster, they may be merged into a single associative node with several children; the node's children may then be rearranged in any order conducive to DAG or code reduction (e.g., constant subsumption or register allocation.)

Should an associative transformation be applied during or after the DAG construction? We don't know. If it is applied during construction, then an expression C+B+A is merged into a cluster; this merger will then prevent recognition of A+B, B+C, etc., later. If it is applied after construction, then we need to search the DAG for equivalent, or partially equivalent clusters,

e.g., B+C+A will be one cluster and C+A+B will be another one; the two could be merged, but will require some searching. On the whole, there is no evidence that an associative transformation in a DAG is worthwhile as a means of detecting common subexpressions. It is certainly worthwhile for the subsumption of constants and for improving register allocation.

11.4.8. DAG Reduction

All of the temporary variables may be eliminated from the DAG, once built, which will incidentally also remove any redundant operations. A node may still carry more than one output variable. If it does, we have a situation in which a single variable (i.e., one definitional value) is represented by more than one name. A common name can clearly be assigned to these variables.

Finally, any root node that does not contain an output variable name may be eliminated. Such a node represents a definition of a temporary variable that is never referenced.

Example

Consider the block

$$
\begin{aligned}
V &\leftarrow A*B \\
T &\leftarrow A+C \\
C &\leftarrow B*D \\
X &\leftarrow V+C \\
Z &\leftarrow T-C \\
C &\leftarrow V+C
\end{aligned}
$$

with the input set {A,B,C,D} and output set {C,V,Z}.

An equivalent normal block is

$$
\begin{aligned}
V &\leftarrow A*B \\
T &\leftarrow A+C'' \\
C' &\leftarrow B*D \\
X &\leftarrow V+C' \\
Z &\leftarrow T-C' \\
C &\leftarrow V+C'
\end{aligned}
$$

with the input set {A,B,C'',D} and the output set {C,V,Z}.

The evaluation of V+C' is clearly redundant; the redundancy is detected during the DAG construction. The final DAG is shown in figure 11.12. We also see that X is useless; it appears in a root node, but is a temporary variable; it is associated with node n_4. The local variables T and C' are associated with nodes n_2 and n_3, respectively.

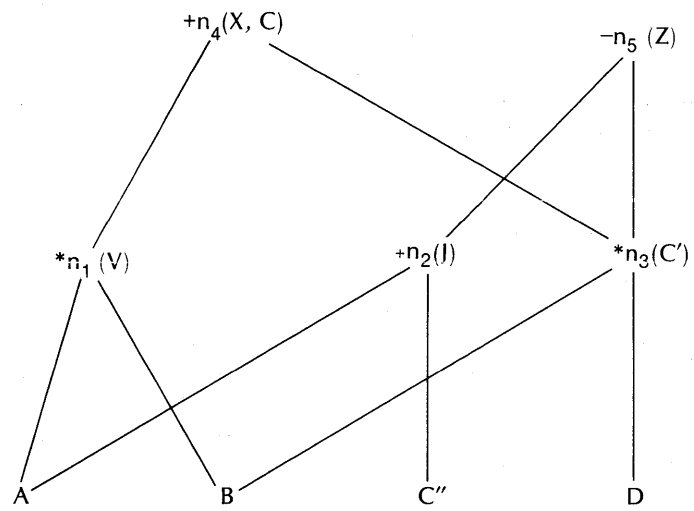


Figure 11.12. A DAG for the block {V ← A*B, T ← A+C″, C′ ← B*D, X ← V+C′, Z ← T−C′, C ← V+C′}.

11.4.9. DAG Evaluation

Let us assume binary operations only in the following discussion, for the sake of simplicity. (This restriction is not hard to remove.) Also let us assume that a multiregister machine as defined in section 11.3 is to be used for evaluation of a DAG. This machine, it will be recalled, has LOAD and STOR operations between a memory location and a register, and binary operations between two registers or between a register and memory, the result going to a register.

We would like an algorithm that, given any DAG, will generate optimal code. Unfortunately, such an optimization is a large combinatorial problem—the computational complexity of any optimal algorithm increases exponentially with the number of nodes (Aho [1977]). Even simpler target machines, e.g., machines with one register or with an unlimited number of registers, have no optimal algorithm of less than exponential complexity.

However, there are several heuristic algorithms for single and multiregister machines; these algorithms are reviewed in Aho [1977]. Let us examine one of them in some detail, suggested by Waite [1974b], called the *top-down greedy* (TDG) algorithm.

The TDG algorithm builds a list L of internal nodes. This list, when used in reverse, will yield a reasonably good node evaluation sequence for a multiregister machine of the class described in section 11.3, containing NR registers.

```
procedure TDGP(N, K: integer);
   {N is a DAG node, K is the number of
        available registers}
if (K ≤ 1) and (N is an internal node
        all of whose parents
        are on the list L) then
begin
   Add (N) to list L;
   TDGP(LEFTCHILD(N),K − 1)
   TDGP(RIGHTCHILD(N),K);
end;

  {Now the main program}

while (not all internal nodes are
        on the list L) do
   (select an internal node N, all of
    whose parents are on L, and
    perform TDGP(N, NR));
```

The TDG algorithm begins by selecting some root node of the DAG (no parents). Then TDGP will place it on the list L. TDGP will then pursue the left branches of its current node, adding them to the list until some node is found with a parent not on the list, or until no more registers are available. It then pursues the right branches in the same way.

The selection of a node could be influenced by some additional tests. Given a set of possible nodes (i.e., none are on the list and all have no ancestors), choose a node n whose left descendant n′ is such that n is the only ancestor of n′ not on the list. If there is more than one such n, look at the left descendants of n′, etc. This strategy will yield a sequence of DAG nodes that tend to be linked together; the number of temporary memory references will thereby be minimized to the extent possible under such a simple approach.

Example

Consider a machine with NR=2 (two registers) and the DAG of figure 11.13, with input variables A, B, C, D and output variables X and Y.

TDG yields the list

7, 5, 6, 3, 4, 1, 2

We then use this list in reverse, along with a reasonable register allocation scheme (one is described later), to obtain the code sequence given below. The following table indicates the node or variable currently in one of the two registers {R1, R2}.

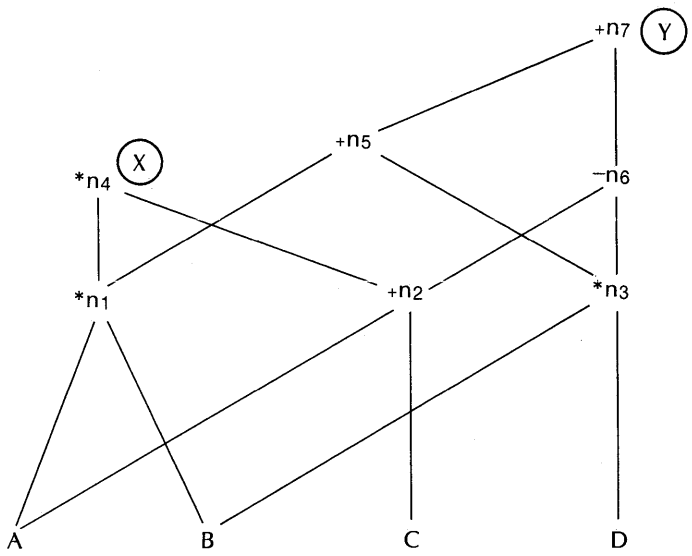


Figure 11.13. An example DAG (see text).

	contents of R1	R2
R1 ← A	A	
R2 ← R1+C		n_2
R1 ← R1*B	n_1	
T1 ← R1 {temporary}	–	
R1 ← R1*R2	n_4	
X ← R1		
R1 ← B	B	
R1 ← R1*D	n_3	
R2 ← R2−R3		n_6
T6 ← R2 {temporary}		–
R2 ← T1		n_1
R1 ← R2+R1	n_5	
R1 ← R1+T6	n_7	
Y ← R1		

Exercises

1. Construct a DAG for the block

```
X := A*(B+C);
Y := (B+C)/A;
X := X-A;
B := X/Y;
Y := A+B+C;
```

X and Y are the output variables; A, B, and C are input variables.

2. Show that a TDG list for the DAG of figure 11.12 is:

5, 2, 4, 1, 3

Develop a code sequence for the list of exercise 2, for NR=2. Can you find a shorter sequence for some other list?

4. Let the reverse of a TDG list be $n_1, n_2, \ldots, n_r$, and let the nodes be evaluated in that order. Show that when any node N is to be evaluated, all of its children (if any) have been previously evaluated.

11.4.10. Register Assignment and Code Generation

Suppose that we are given an ordering of the internal nodes such as that provided by the reverse of the list generated by TDG. We may then assign registers and emit code by a heuristic algorithm similar to the following.

1. We maintain a list of assignments of registers to nodes. A register R is assigned to a node N if, at that point in the code sequence, node N's value is carried by register R. Initially, no register is assigned to any node. A register is said to be *available* if it is not assigned to a node. We also assign a level number to each node in the DAG; initially, this will be the number of parents of the node. As operations are coded, the level numbers will be reduced to 0.

2. Then the following procedure EMITNODE is executed for each node N on the node list generated by TDG or some other algorithm:

```
procedure EMITNODE(N);
    {N is a DAG node, all of whose
       children have been evaluated}
begin
   if (LEFTCHILD(N) value is held in register R) then
   begin
      if LEVEL(LEFTCHILD(N))=1 then
      begin {a level of 1 means that after this use, the
               left child's register will be available}
```

```
            EMITOP(OP(N),R,VALUE(RIGHTCHILD(N)),R);
              {the right child may or may not be in a register;
                  either instruction form may be emitted}
            ASSIGN(R,N)    {assign R to node N}
          end
        else
          begin {left child will be needed after this operation}
            var b': integer;  {declare a temporary}
            R' := ALLOCATE;  {get a register; it could be R}
            EMITOP(OP(N),R,RIGHTCHILD(N),R');
            ASSIGN(R',N)
          end
      end
    else
      begin {left child is not in a register}
        var R: integer;
        R := ALLOCATE;  {allocate a register}
        EMITLOAD(VALUE(LEFTCHILD(N)), R);
            {load leftchild in register R}
        ASSIGN(R,LEFTCHILD(N));
        EMITNODE(N);  {try again; left child is now in
              a register}
        return    {cleanup was done on previous call}
      end;
    {begin some cleanup for node N}
      if (N carries an output variable X) then
        EMITSTOR(X, REGISTER(N));  {emit a store
            to X from the register assigned to N}
      DECRLEVEL(N)  {adjust levels and release registers}
  end;
  procedure DECRLEVEL(N: integer);
  begin
    {this decrements the level numbers of the
     children of node N.  If the level of N or any of
     its children is then 0, the corresponding registers
     are released}
    var NLC, NRC: integer;  {temporary node numbers}

    NLC:=LEFTCHILD(N);
    NRC:=RIGHTCHILD(N);
    LEVEL(NLC):=LEVEL(NLC)-1;
    LEVEL(NRC):=LEVEL(NRC)-1;
    if LEVEL(NLC)=0 then RELEASE(NLC);
      {if a register or temporary is not assigned to node N,
```

```
        RELEASE(N) does nothing; otherwise it makes
        the register or temporary assigned to N available}
  if LEVEL(NRC)=0 then RELEASE(NRC);
  if LEVEL(N)=0 then RELEASE(N)
end;

procedure ALLOCATE: integer;
begin  {allocates a register, returning its number}
  if (any register R is available) then return(R)
 else
  begin
    var T: integer;
    T:=ALLOCATETEMP;  {allocate temporary mem-
                       ory cell}
    if (any register R is assigned to a node N such
        that N is not a left descendant of any of
        its ancestors) then
    EMITSTOR(R, T)    {save R in cell T}
   else
    begin  {no reasonable basis for choice}
      Choose any register R assigned to a node N;
      EMITSTOR(R, T)    {save R in cell T}
    end;
    ASSIGN(T,N);  {temporary T assigned to node N}
    return(R)
  end
end
```

These procedures exploit several possibilities that can arise. If a register is needed and one is available, then it is used. If a register is needed and none are available, then we look for one assigned to a node that is only a right descendant of its ancestors; such a node can later be accessed directly through its temporary memory cell and need not be allocated a register. Barring these possibilities, there is probably no good basis for a register choice.

The register allocator is responsible for saving the present contents of an assigned register; that value will be needed later, since the register cannot be assigned to a node at the end of an EMITNODE call if the level of the node has fallen to zero.

Exercises

1. Generate a code sequence for the DAGs of Figs. 11.12 and 11.13 using TDG and EMITNODE.
2. Show that EMITNODE emits correct code for a DAG, given a node list with the evaluation order property that a node is selected for evaluation only if each of its children has been evaluated.

3. Discuss extensions to TDG and EMITNODE for n-ary operators, e.g., a function call, a MAX operator, etc. Can an n-ary operator be transformed into a sequence of binary operators?
4. Devise data structures for a block, a DAG, and whatever else is needed to implement a code generation system. Write Pascal procedures that emit code for a given block, expressed as a sequence of binary assignment operations.
5. Arbitrary choices are possible in both ALLOCATE and TDG. Where? Suppose, as an optimizing strategy, that these choices are made through a backtracking, nondeterministic machine. We can then generate all possible equivalent code sequences and choose the shortest one. Show that such a system must halt in finite time. Is this computationally feasible? How would the number of operations vary with the size of the DAG? (A measure of the size of a DAG might be the number of nodes).

11.5. Data Flow Analysis

In considering optimization of blocks, we need information regarding the definition and use of data items that cannot be found by an examination of the block alone. For example, we cannot distinguish output and temporary data items; we need to know whether any reference to some data item will lie in a control path leading from the block, uninterrupted by another definition. In short, we cannot determine the domain of the block variables by information in the block alone.

If we had a complete data flow analysis of a program, we could provide, in addition to block optimization, some useful tests and other optimizations as follows:

1. A sneak path, along which a reference to some data item is not preceded by a definition, can be detected. The programmer should be made aware of such paths, although the logic of the program may be such that the path can never be completely followed to the reference point.
2. Redundant or useless code can be detected. A redundant assignment statement is one that is followed eventually by an identical one. A useless assignment is one for which no subsequent reference exists.
3. Redundant or useless variables can often be detected. A redundant variable is one for which another variable always carries the same value. A useless variable is one for which only definitions and no references exist. A redundant variable can only be detected under certain circumstances; in general, it is not possible to determine from a static analysis of a program whether two variables carry identical values.

4. Register allocations might be carried from one block into others, or through a block, if it appears that some code savings might result.
5. Loop invariant variables can often be identified.

Data flow analysis is based upon a *control flow graph*. Given a program, its flow graph G consists of a connected, directed graph (not necessarily acyclic) having a single entry node n_0. Graph G consists of a set of nodes $N = \{n_1, n_2, \ldots, n_m\}$, representing blocks of instructions, and a set of edges E, connecting pairs of nodes, that represent the branching paths between blocks.

Each block has one entry and one exit point and consists of the largest sequence of program steps that can be so formed. A block might also consist of the instructions of a set of blocks connected together in such a manner as to contain only one entry and one exit point; we call such a block an *extended block*.

The detailed nature of a branch decision is ignored in data flow analysis, as are the details of the calculations within a block. The only information of interest is (1) the possible branches, (2) the definitions of data items, and (3) the uses of data items. It is entirely possible that the branching decisions in a program are such that some paths can never be followed during execution; however, for our purposes, we must assume that any directed path can be followed during execution.

11.5.1. Definitions

A *definition* of a data item R is some statement that assigns a value to R, replacing a previous assignment. A *use* of R is some statement that requires the current value of R in a computation.

A *locally available definition* of a data item R in a block B_i is the last definition of R appearing in that block. We denote the set of available definitions for block B_i by DB_i.

Any definition of an item R is said to *kill* all definitions of R that can reach the block containing the definition. Any definition of a data item R that can reach a block are *preserved* by the block if R is not defined in the block. The set of all definitions in the program that are preserved in block B_i will be denoted PB_i.

A definition d in a block B_1 is said to *reach* block B_2 if both of these conditions hold:

1. Definition d is locally available from B_1,
2. Definition d is preserved on some directed path P from B_1 to B_2(but not necessarily on all paths.) Path P may be null.

The notion of *reaching* is essentially that the value of a data item X assigned in block B_1 can somehow get through to B_2, without an intervening redefinition of X. The set of definitions that reach a block B_i will be denoted R_i.

A definition in a block B is said to be *available* at block B′ if it is locally available in B and can reach B′. We denote the set of available definitions at a block B_i by A_i.

A *locally exposed use* of a data item X in a block B is a use of X in B that is not preceded in the block by a definition of X. A variable with a locally exposed use in block B is an input variable of B.

A use of a data item X is *upwards exposed* in block B if it either is locally exposed in B, or if there exists a path through B and to some other block B′ that does not contain a definition of X, and such that X is locally exposed in B′. Here the notion is that some use of X in block B′ can be the terminus of some path running through a number of blocks, including a block B; that use is then upwards exposed at B. The set of upwards exposed uses of a block B_i will be expressed as a set U_i. Note that this set contains uses that appear in block B and elsewhere in the program.

A definition d is *active* or *live* at block B_i if d reaches B_i (i.e., $d \in R_i$) and there is an upwards exposed use at $B_i$of the data item defined by d. The set of live definitions at block $B_i$will be denoted L_i. Clearly,

$$L_i = R_i \cap U_i$$

The set intersection of R_i and U_i corresponds to the two requirements that there be an upwards exposed use and that definition d reach block B_i.

Examples.

Consider the following block:

```
K:=3;     {1}
J:=3*K;   {2}
N:=J-2;   {3}
J:=3*N;   {4}
N:=J+6;   {5}
```

The *locally available* definitions out of this block are those for J, K, and N, lines 1, 4, and 5. The definitions in lines 2 and 3 are *killed* by the subsequent lines 4 and 5.

There are no locally exposed uses for this block; these uses would be input variables, and there are none. If the first statement were

```
K:=J+N;
```

instead, then these uses of J and N would be locally exposed.

Now consider the following block:

```
K:=17;    {1}
J:=M+3;   {2}
K:=J-M;   {3}
```

A definition of M preceding this block is preserved by the block; there are no definitions of M in it. If the program contains definitions for variables {A, B, J, K, M}, then this block also preserves any definitions for A and B, but not for J and K.

Consider figure 11.14. A definition d (the assignment statement K:=17) appears in block B_1. There are no subsequent definitions of data item K in blocks B_1 or B_2. We then say that definition d *reaches* block B_3; whether it is killed in block B_3 is immaterial. There may also be other paths from B_1 to B_3 along which definition d is killed; we only need one along which the definition is preserved for d to reach B_3. Block B_1 may also be the same as block B_3; we then have a loop through which a definition in block B_1 can reach its own block.

The definition d in block B_1 in figure 11.14 is locally available in B_1 and is available in blocks B_2 and B_3.

Consider figure 11.15. A use u of K appears in block B_3, through the statement M:=K+3. If no definition of K appears prior to u in block B_3 or anywhere in blocks B_1 and B_2, then use u is upwards exposed in all three of these blocks.

Next, consider figure 11.16. We have a definition d of data item K in block B_1, with no subsequent definition in B_1, B_2, B_3, or B_4. A use of K appears in B_4. We then say that d is live in blocks B_2, B_3, and B_4. d has clearly reached each of these blocks, and an upwards exposed use of K exists in each of them. Again, the existence of other paths or of definitions in other blocks not shown in the figure is immaterial.

Finally, consider figure 11.17. We have several blocks, containing three definitions and two uses of a variable X. All the control flow paths are shown. We denote a definition of a data item X in a block i by X_i. Sets R_j and L_j for a block j will consist of definitions. Set U_j for block j will consist of the one data item X or be empty.

For example, consider block 3. Only the definition X_2 can reach this block; the definitions in blocks 6 and 7 cannot. Hence R_3 consists of X_2. Set U_3 contains X, since a use of X appears in block 3 not preceded by a definition. Finally, definition X_2 is live in block 3; none of the other definitions are.

A complete table of reaches, upward exposed uses, and live definitions for the graph of figure 11.17 is given below.

Block i	Reach R_i	Up. Exp. U_i	Live Defs. L_i
1	∅	∅	∅
2	X_2	∅	∅
3	X_2	X	X_2
4	X_2	X	X_2
5	X_2	X	X_2
6	X_2	∅	∅
7	X_2, X_6	∅	∅

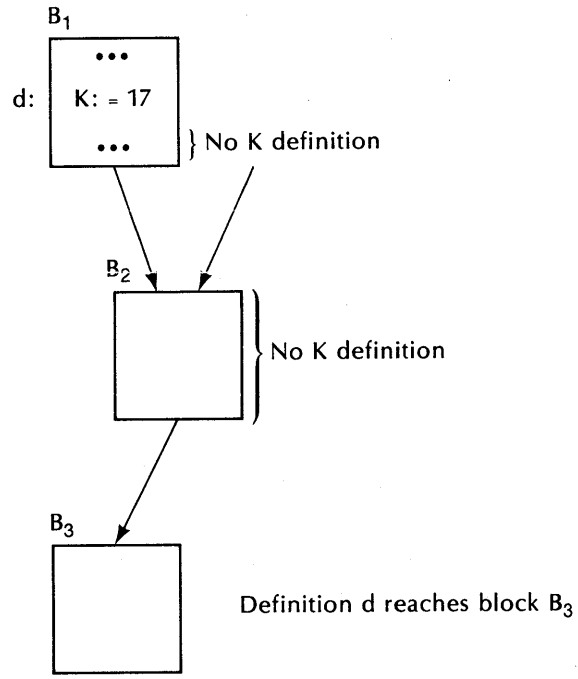


Figure 11.14. *Reach* of a definition d to block B_3.

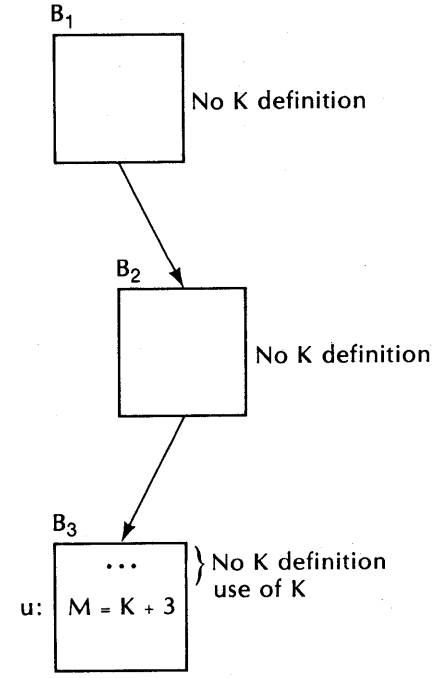


Figure 11.15. *Use* u of K in B_3 is *upwards exposed* in all three blocks B_1, B_2, B_3.

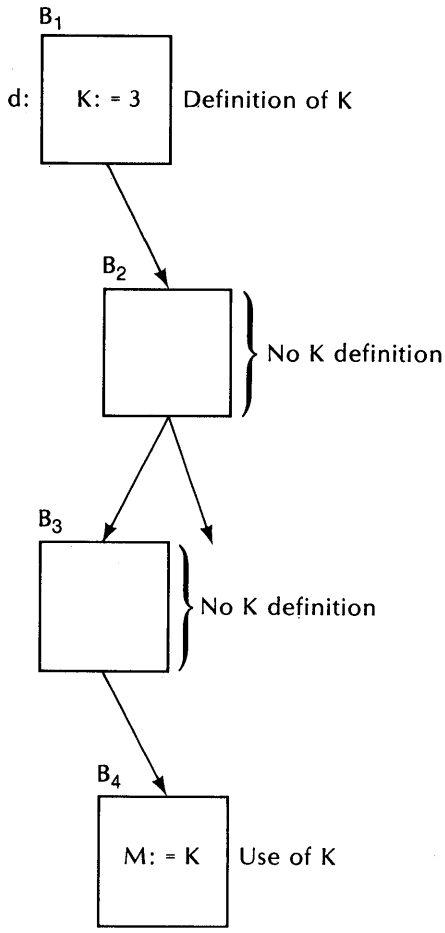


Figure 11.16. *Definition* of K in block B_1 is *live* in blocks B_2, B_3, B_4.

Exercises

1. Consider the graph of figure 11.18, consisting of 7 blocks. There are definitions of a data item X in blocks 5 and 7, and uses in blocks 2 and 4. Determine the sets R_i, U_i, and L_i for each block i, $1 \leq i \leq 7$.
2. A Basic program carries a line number on each statement. If such a program contains a computed GOTO statement of the form GO N, where N has been previously evaluated and must evaluate to a line number during program execution, is it possible to construct a data flow graph for the program? What form would it have?Similarly, consider a

Fortran program with a computed GOTO, but such that only some of the statements carry line numbers.

3. Construct a flow graph for the statements of the following program and construct the sets R, U, and L:

```
for G=1 until NOOFGRAPHS-1 do
begin
   if G>1 then
   begin
        for I=FIRSTEDGE(G) until LASTEDGE(G) do
        begin
```

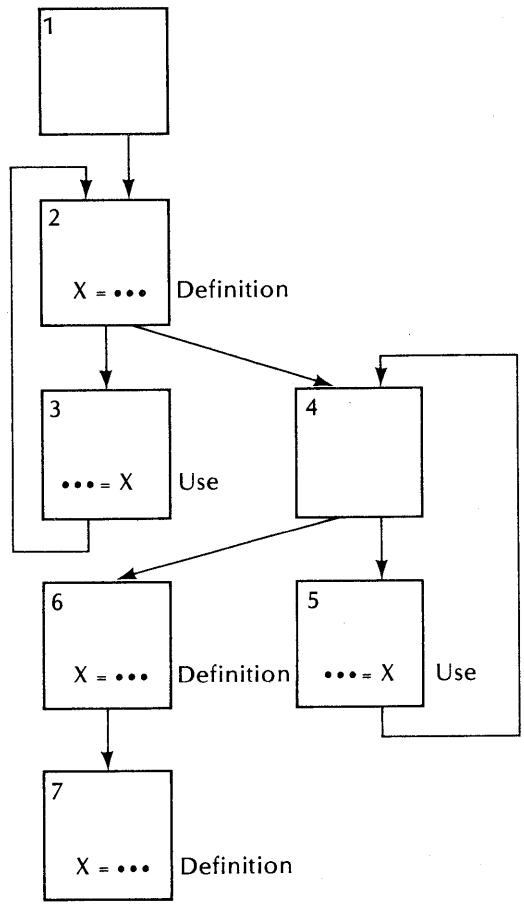


Figure 11.17. A complete flow graph with some definitions and uses of variable X.

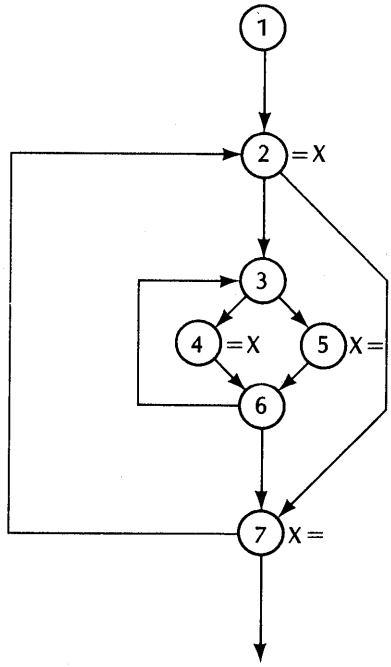


Figure 11.18. An example graph—see exercise 1.

```
            PB(I) := P(CORRESEDGE(I));
            DB(I) := REACH(CORRESHEAD(FROM(I)))
                  and PB(I)
                  or D(CORRESEDGE(I));
         end;
         for I:=FIRSTNODE(G) until LASTNODE(G) do
         begin
            UB(I):=USED(CORRESHEAD(I));
            REACH(I):=REACH(CORRESHEAD(I));
         end
   end
end;
HEAD:=0;
```

(*Note:* There are no function calls in this program; all the subscripted references are vector references).

11.5.2. The Basic Data Flow Analysis Method

It should be apparent that the determination of the various sets defined here

can be made by first determining the local sets PB and DB for each block, then exploring all the paths that lead into or out of each of the blocks and the implications of PB and DB on these paths. Unfortunately, path exploration in a directed graph is often a computationally hard problem—the number of operations tends to increase exponentially with the number of nodes in the graph. It turns out that data flow analysis is not computationally hard, provided that a graph reduction technique called *interval ordering* is used. Interval ordering reduces the computational difficulty of flow analysis to one requiring O(n log n) operations, where n is the number of nodes.

Let us first examine the basic method. It should be apparent that the set of definitions that reach a block B_i is the union of the definitions available from the nodes that are the immediate predecessors (the parents) of B_i. That is,

$$R_i = \cup A_p$$

where the A_p are the available definitions of the blocks $B_p$that are the parents of block B_i.

Now the set of available definitions A_i for block B_i consists of all those locally available definitions DB_i, together with all those definitions in the program that reach B_i (namely R_i) and that are preserved in B_i. That is,

$$A_i = (R_i \cap PB_i) \cup DB_i$$

These two set equations are sufficient to determine the sets A_i and R_i for all the blocks B_i through a simple algorithm called the *basic reach algorithm*:

Basic Reach Algorithm

The inputs are the PB_i and DB_i for each block B_i in the control flow graph. The outputs are R_i and $A_i$for each block B_i.

1. Initialize all of the A_i and R_i to the null set.
2. Perform step 3 while there is no change in any R_i or A_i.
3. Apply the following two formulas in succession to the nodes of the graph:

   $$R_i = \cup A_p$$

   where the A_p are the available definitions of the blocks $B_p$that are the parents of block B_i.

   $$A_i = (R_i \cap PB_i) \cup DB_i.$$

Kildall [1973] has shown that the information in A_i and R_i does stabilize. However, the rate with which the sets stabilize depends critically on the order in which the nodes of the graph are examined; the process can be very time consuming for a poor choice of node ordering.

Note that, to be useful, we must consider a program at least as large as a typical procedure, and build definition and preservation sets for all its definitions and uses. We cannot arbitrarily restrict our interest to some subset of the program. It is therefore important to develop an efficient method of determining these sets.

11.5.3. Intervals

Given a node h, an *interval I(h)* is the maximal, single entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the *interval head* or simply the *header node*. An interval will consist of an ordered list of nodes; the ordering will be determined by the next algorithm, and will be important in an improved reach algorithm.

Interval Algorithm

1. Let H be a set of header nodes. Initially, let it contain node n_0, the unique entry node of the flow graph.
2. For every node h ϵ H, fix the interval list I(h) as follows:
   (a) Put h in I(h) as the first element.
   (b) Add to I(h) any node all of whose immediate predecessors are already in I(h).
   (c) Repeat step 2(b) until no more nodes can be added to I(h).
3. Add to H all nodes in G that are not already in H and that are not in I(h), but that have immediate predecessors in I(h). Clearly, a node is added to H the first time any, but not all, of its parents become members of an interval.
4. Select the next unprocessed node in H and repeat steps 2, 3. If no more unprocessed nodes exist in H, the procedure terminates.

The end result is a list of intervals $I(h_1)$, $I(h_2)$, . . . , that represents a partition of the nodes of the flow graph.

Example.

Figure 11.19 illustrates the partitioning of a graph into intervals. Interval I(1) consists of only one node 1, since its successor, node 2, has more than node 1 as a parent; there is a path from node 7 to 2 as well as from node 1, and node 7 is not in I(1). Interval I(2) is begun with node 2, and again consists only of that node. (Node 3 can be reached from node 6.) Interval I(3) is begun with node 3, then nodes 4 and 5 can be added, then 6. Node 7 cannot be added, since a path from node 2, not in I(3), exists. Finally, interval I(7) consists of nodes 7 and 8.

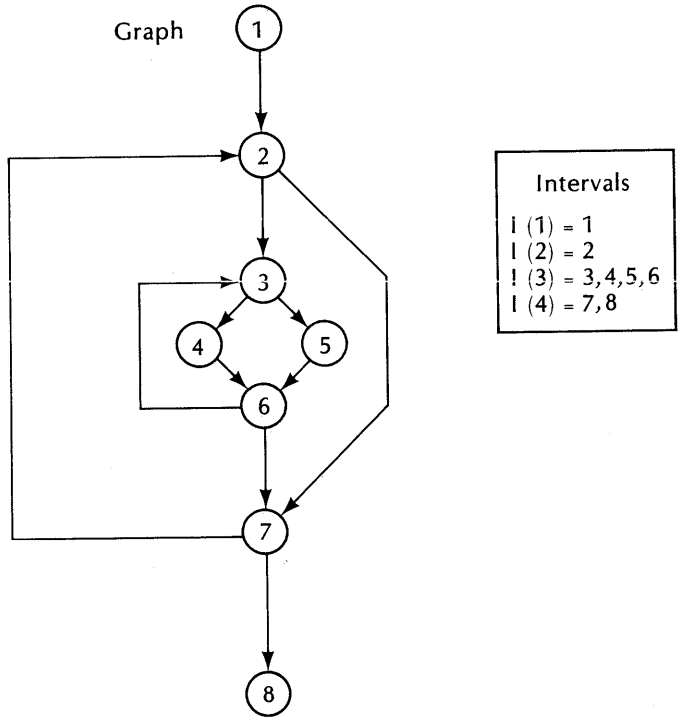


Figure 11.19. Partitioning a graph into intervals.

11.5.4. Higher order intervals

The intervals just determined from the original flow graph G^1 can be expressed as another graph G^2. We construct G^2 from the intervals of G^1 by merging the nodes within each interval, and discarding all edges connecting two nodes in the same interval. We thus obtain a *second-order interval* graph. This, too, can be reduced by interval partitioning to obtain a third-order, a fourth-order, etc. interval graph.

Eventually, a graph G^i is obtained by interval analysis of a graph G^{i-1} that contains the same number of nodes as G^i. If G^icontains one node, we say that G^i is *reducible;* otherwise G^i is *irreducible.*

An example of an irreducible graph is given in figure 11.20. Here, I(1) consists of 1 alone, I(2) consists of 2 alone, etc.

It is interesting that programs that consist only of structured control statements (if-then-else, while-do, for, sequential statements) are reducible. Irreducible graphs stem from programs with GOTO statements.

An irreducible graph can be *split* by replicating one or more nodes. For example, if nodes 3 and 4 are duplicated in figure 11.20, we obtain a reducible graph, figure 11.21. Splitting is merely an analysis technique; we are not suggesting that the program code must be duplicated. It also turns out that

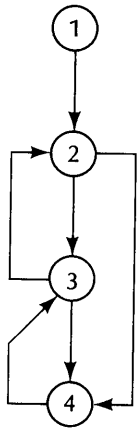


Figure 11.20. An irreducible graph.

even if a control flow graph is irreducible, the analysis that yields sets R and A nevertheless terminates and correctly computes the sets. It will just take more time than a reducible graph with a comparable number of nodes.

The details of the reach algorithm based on intervals may be found in Allen and Cocke's paper (Allen [1976]). The general reach algorithm is in two phases. In the first phase, two items of information are collected by working through the interval graphs in the order G^1, G^2, . . . (the order in which they were constructed) as follows:

1. The definitions defined in the interval and locally available from it. They will become a DB set for the node representing the interval in the next higher order graph.
2. The definitions preserved by the interval; they will become the PB sets for the next higher order graph.

In the second phase, the graphs are worked through in the opposite order of construction; the set of definitions reaching each node is thereby generated.

The essential property of an interval that reduces the length of the analysis is that control that reaches any node within the interval must have passed through the interval header. We say that the header *dominates* the nodes of the interval for this reason. Furthermore, an interval I′ that is the successor of an interval I is such that control must pass from some node in I into the header of I′. By considering the interval sets in their generation order, the number of path combinations to be considered are minimized.

It is interesting that Knuth, in his empirical examination of a number of Fortran programs (Knuth [1971]), found that over 90 percent of their control flow graphs were reducible, which implies that a control flow analysis based

on intervals will generally be quite efficient. An analysis of programs written in a more structured language, such as Algol or Pascal would undoubtedly yield an even higher percentage of reducible control flow graphs. An Algol or Pascal program without GOTO statements would be reducible.

11.5.5. Use and Live Information

Given the upwards exposed use U_i and the definitions R_i that reach a block B_i in a program, the live set can readily be determined. However, this set contains only the data items that are live upon entry into a block. It is usually more desirable to know those items that are live on each of the exit edges from a block. For example, register allocation requires live information on exit from a block, rather than on entrance. Rather than retain both entrance and exit information, it is better to retain live information for edges and construct the entry live data item sets as needed; the latter is the union of the former over the entering edges.

We then need the set of definitions A_e available on each edge. Then for a given edge e, incident into node i,

$$L_e = A_e \cap U_i$$

The algorithm for generating the U_i sets for the graph nodes by the use of the interval sets again requires two phases, the first in interval generation order and the second in opposite order. The first phase can be a part of the reach algorithm. The second phase, however, works backwards through the nodes in each interval.

The upwards exposed uses from each interval is found in the first phase as follows:

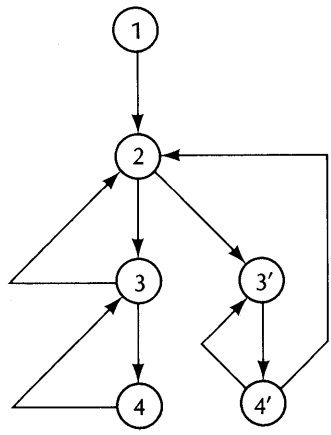


Figure 11.21. Splitting states 3 and 4 in the graph of figure 11.20.

1. Prior to processing each interval h, a set U_h is created and initialized to the upwards exposed uses in that interval.
2. For each node i in the interval (i = 2, 3, . . .), U_h is updated with the locally exposed uses UB_i in i that can be preserved along some path from h to i, which is done by forming the set

$$(\cup P_p) \cap UB_i$$

where the union is over the sets P_p; P_p is the set of data items preserved on input edge p to node i. Its computation is part of the basic reach algorithm.

The complete algorithms and a procedure in PL/I may be found in Allen [1976].

11.5.6. The Interval-Based Reach Algorithm

Inputs

1. The ordered set of graphs (G^1, G^2, . . . , G^n) determined by interval analysis.
2. The intervals in each graph with their nodes given in interval process order.
3. The definitions *defined* and *preserved* on each edge in the first order graph. These are expressed in the DB and PB sets.

Outputs

1. A set R of the definitions that reach each node.
2. A set A of the definitions available on each edge.

Phase I

1. For each graphG^g in the order G^1, G^2, . . . , G^{n-1}, perform steps 2 and 3 that follow:

2. If the current graph is not G^1 then initialize the PB and DB sets for the edges of the graph. PB and DB are initialized by first identifying the edge in G^{g-1} to which each edge in G^g corresponds (these will be interval exit edges). Then, using the information generated during step 3 for G^{g-1}, for each edge i in G^g with corresponding exit edge x from interval with head h in G^{g-1}, set:
(a) $PB_i := P_x$, and
(b) $DB_i := (R_h \cap P_x) \cup D_x$

3. For each exit edge of each interval in G^g, determine P, the definitions preserved on some path through the interval to the exit, and D, the definitions in the interval that may be available on the exit. These definitions are found by finding P and D for each edge in the interval as follows:

(a) For each exit edge i of the header node:

$$P_i := PB_i$$

$$D_i := DB_i$$

(b) For each exit edge i of each node j (j = 2, 3, . . .) in interval order:

$$P_i := (\cup\ P_p) \cap PB_i$$

$$D_i := ((\cup\ D_p) \cap PB_i) \cup DB_i$$

where the union over P_p and D_p is for all p input edges to node j.

While processing an interval, determine the set of definitions R_h that can reach the interval head h, from inside the interval by

$$R_h := \cup\ D_l$$

for all interval edges l which enter h. If there are none set R_h to $\varnothing$.

Between phases I and II, the R vector for the single node in the n'th order derived graph is initiated: $R_1 = \varnothing$ or whatever set of definitions is known to reach the program from outside.

Phase II

1. For each graph G^g in the order $G^{n-1}, \ldots, G^2, G^1$, perform the following steps 2 and 3.

2. For each node i in G^{g+1} form $R_h := R_h \cup R_i$, where h is the head of the interval in G^g which i represents in G^{g+1}.

3. For each interval process the nodes in interval order determining the definitions reaching each node and available on each node exit edge as follows:

(a) For each exit edge i of the header node h

$$A_i := (R_h \cap PB_i) \cup DB_i$$

(b) For each node j (j=2, 3, . . .) in interval order first form

$$R_j := \cup\ A_p, \text{ for all input edges p to j}$$

then for each exit edge i of j form

$$A_i := (R_j \cap PB_i) \cup DB_i$$

Exercise

Consider the graph G^1 and its higher order interval graphs shown in figure 11.22. There are two variables, X and Y, and three definitions X_1, X_4, and Y_3.

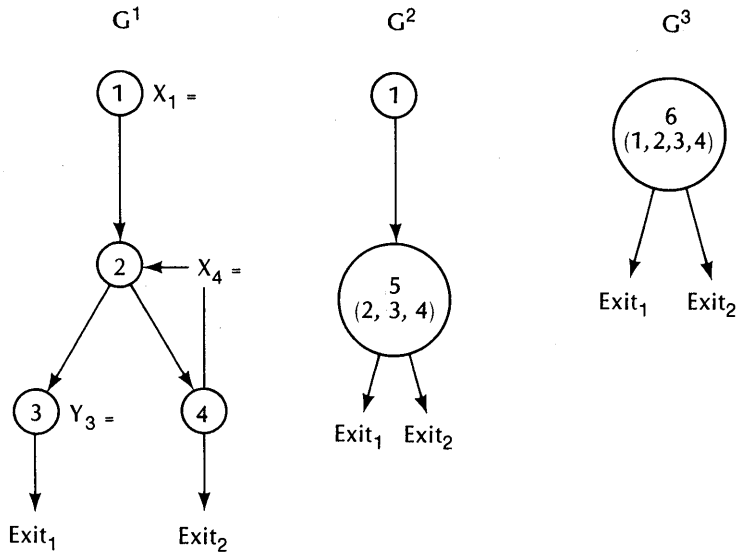


Figure 11.22. A graph and its higher-order intervals.

(a) Show that figure 11.23 expresses DB and PB for the graphs.

(b) Show that figure 11.24 expresses the reach and availability sets R and A after the second phase of the reach algorithm.

	Edge	DB	PB
G^1	1-2	X_1	$X_4\ Y_3$
	2-3	0	$X_1\ X_4\ Y_3$
	2-4	0	$X_1\ X_4\ Y_3$
	3-e_1	Y_3	$X_1\ X_4$
	4-2	X_4	$X_1\ Y_3$
	4-e_2	X_4	$X_1\ Y_3$
G^2	1-5	X_1	$X_4\ Y_3$
	5-e_1	$X_4\ Y_3$	$X_1\ X_4$
	5-e_2	X_4	Y_3
G^3	6-e_1	$X_1\ X_4\ Y_3$	$\emptyset$
	6-e_2	$X_1\ X_4$	Y_3

Figure 11.23. DB and PB for the graph of figure 11.22.

	Node	Reach	Edge	Available
G^3	6	$\emptyset$	$6\text{-}e_1$	$X_1\ X_4\ Y_3$
			$6\text{-}e_2$	X_4
G^2	1	$\emptyset$	1-5	X_1
	5	X_1	$5\text{-}e_1$	$X_1\ X_4\ Y_3$
			$5\text{-}e_2$	X_4
G^1	1	$\emptyset$	1-2	X_1
	2	$X_1\ X_4$	2-3	$X_1\ X_4$
	3	$X_1\ X_1$	2-4	$X_1\ X_4$
	4	$X_1\ X_4$	$3\text{-}e_1$	$X_1\ X_4\ Y_3$
			4-2	$X_1\ X_4$
			$4\text{-}e_2$	$X_1\ X_4$

Figure 11.24. REACH and AVAILABILITY sets for the graph of figure 11.22.

11.5.7. Applications of Data Flow Information

We are now in a position to outline a number of useful optimizations based on the information developed in a data flow analysis.

Useless definitions. A definition d is *useless* if the value assigned to the data variable X in d is never subsequently required by any reference. The definition d may be removed with no effect on the program. Let d reside in a block B_i. Now d will be useless if:

1. Definition d is live on the B_i block exit, i.e., d is in L_i.
2. No upwards exposed use of d exists on exist from B_i.

Let U′ be the union of the sets L_e for each edge e leaving block B_i. If d is not in U′, then d is useless. Set U′ is a *useful* set of definitions.

Uninitialized Variable Use. If a use u is upwards exposed in the "start" block B_0, then a path from the program origin to u exists that does not contain a definition of the use data variable. Clearly, a test for potentially uninitialized variables in a program is that $U_0 = \emptyset$. We say "potentially", because data flow analysis does not consider the logic of program branching, and it may happen that the path to the uninitialized variable use cannot be followed during program execution.

Basic Block Input and Output Variables. An input variable X of a block B is such that a use u of X appears in B, and u is upwards exposed in B. As we have seen earlier, we do not need a data flow analysis to identify input variables of a basic block.

An output variable X of a basic block B must appear in B (use or definition)

and is live on exit from B. The *live* set of course derives from data flow analysis.

Loop Code Movement. Suppose that a loop is identified, as indicated in figure 11.25. B1 and B2 are some collections of code with single entries and exits; these are not necessarily basic blocks. B3 is a basic block containing a statement S of interest. T is a test for loop exit, and contains only uses. Statement S has the form

$$X \leftarrow \theta\ Y_1\ Y_2 \ldots Y_r$$

where θ is an r-ary operation, the X receives the result, and the Y_i are the operands.

It is sometimes possible to move S from B3 into block B1 without affecting the program. If this can be done, we will have one execution of S in the optimized program for every n executions in the original program, where n is the number of loop iterations. Let us develop necessary and sufficient conditions for such a code movement.

S can be moved to the bottom of B1 only if no reachable definitions of any of the Y_i or X exist in B2, other than in S. Since block B2 is not necessarily basic, we need the results of a data flow analysis.

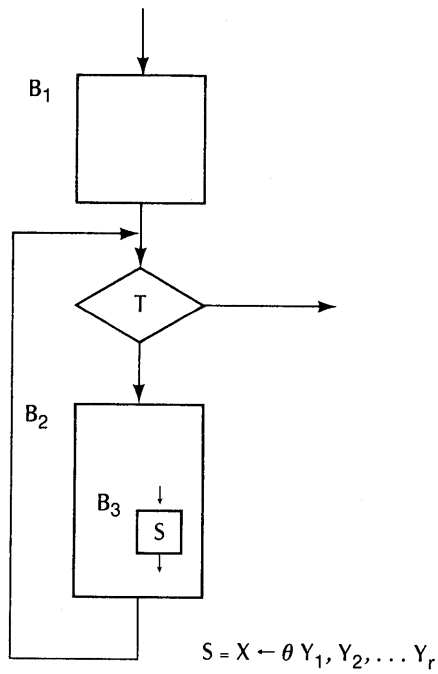


Figure 11.25. Code movement of a statement S from block B_2 to block B_1.

Clearly, all of the uses u_i ($1 \leq i \leq r$) corresponding to the operands Y_i in S must be in the upwards exposed use set U_2 of block B2. They must also be live at the block B2 exit. These conditions guarantee that no loop path can change any of the Y variables.

The definition of X in statement S must be live at block entry B2 (therefore also at exit of B2) and at block entry B3. This requirement clearly precludes any other definitions of X in B2 (except in S and unreachable definitions); such a definition would either kill the S definition at block B2 exit or at block B3 entry.

11.6. Bibliographic Notes

Nakata [1967] and Meyers [1965] developed the optimal algorithm for the generation of code from an expression tree; their algorithm yields the minimum number of registers for commutative operators only. Sethi and Ullman (Sethi [1970]) restated their algorithm, and showed that it is optimal for noncommutative and nonassociative operators as well as commutative and associative-commutative operators.

The notion of collecting complicated nodes to the left of simple nodes for a machine with noncommutative instructions that operate between a register and memory can be traced to an early paper by Floyd [1961].

The discussion of blocks, block statements, and renaming is principally drawn from Aho [1972b]. A presentation similar to ours may be found in Aho and Ullman (Aho [1972]), chapter 11.

The data flow analysis material may be found in Allen and Cocke's paper (Allen [1976]).

Beatty [1974] describes an algorithm that assigns registers in a highly optimal fashion, and he considers both the local domain of a block and its global environment.

Bruno and Sethi (Bruno [1976]) and Aho, Johnson, and Ullman (Aho [1977b]) show that the problem of code optimization for a general class of single and multiregister machines, given a DAG with common subexpression subsumption, is NP-complete, i.e., has a computational complexity that is exponential in the size of the DAG. The significance of this result is that optimal solutions can be demonstrated only for rather small DAGs.

Cocke and Kennedy (Cocke [1977]) has given a simple algorithm that performs reduction of operator strength by moving certain statements out of a FOR loop. A general discussion of loop strength reduction, identification of induction variables, etc. is given in Aho and Ullman (Aho [1972a]), chapter 11.

CHAPTER 12

ERROR RECOVERY

12.1. Introduction

Since the source for a compiler or interpreter is generally prepared by a person, it is likely to contain errors. The compiler must therefore have a reasonable mechanism for dealing with errors. A set of objectives of a satisfactory error recovery system are

1. *Report* the error, indicating its location.
2. *Diagnose* the error, as an aid to its correction.
3. *Recover* from the error such that subsequent errors are detected and spurious error reports are not generated.

We shall see that these objectives can be approached, but not completely met. Objective (1) can always be met, provided that the error has not been masked by the recovery from a preceding error. Objectives (2) and (3) are rather difficult to meet. Effective diagnosis of syntax errors is difficult, and most error recovery systems tend to generate spurious error reports after recovery from some error.

The *correction* of an error, such that the corrected program will execute the programmer's intended algorithm correctly, is not feasible with the commonly used programming languages. It would only be feasible if the program's algorithm were to be expressed in two different, but equivalent ways. With only one statement of the algorithm, and that statement erroneous, a compiler can only patch over the error sufficiently well that it can continue to parse the remaining portion of the program.

The compiler cannot in general determine the error made by the programmer; it sees only an *error symptom*, some condition reached during the parsing process such that the parse cannot continue. An error may or may not immediately produce an error symptom; however, every error symptom is the result of some error. For example, the following Fortran statement contains one too many left parentheses:

```
A = B * (C - (D*F)
                ↑
```

The error symptom is a missing right parenthesis, detected at the end of the line, as indicated by the arrow ↑. However, the error could be earlier in the line; perhaps the programmer intended to write

```
A = B * (C -  D*F)
```

and entered an extra left parenthesis by mistake. The compiler cannot correct the error in general, it can only report an error symptom, and the symptom is often several tokens past the error. We shall often use the term *error* to refer to an *error symptom.*

Error Diagnosis

A reasonable error diagnosis would be an indication of the location of the error, and a brief message indicating the kind of error encountered. Some errors cannot be detected until the end of a block or program; it is especially important that these be clear, and that as much information as possible be provided by the compiler.

The error messages generated during a compilation may be collected and reported at the end of the compilation, or may be interspersed with the source listing. An interspersed listing may be the only possible choice if storage capacity is limited. Otherwise, the choice is a matter of user preference.

If interspersed messages are used, each one should contain a reference to the previous one, and a message at the end of the program should refer to the last message. Otherwise, an isolated error message is easily overlooked or hard to find in a large listing.

An error summary at the end of a listing is important in order for a programmer to quickly determine whether his program is free of errors, and if it is not, to be able to locate each of the errors.

Kinds of Error

There are three major classes of error, depending on whether they are detected in the lexical analyzer (possibly including a macro generator), the parser, or the semantic checking sections of the compiler. We shall call these *scanner, syntax* and *semantic* errors, respectively. A fourth class might be established for a developmental compiler: compiler errors or *bugs,* presumably never seen by an ultimate user, but useful during development to warn of compiler states that should not have occurred.

A *scanner* error results from some problem in identifying a token, or reaching an end of file in the middle of a token, or failure to open or read the source file. A diagnostic message can say nothing about the error's meaning, but must be confined to the specific source problem found. Recovery from a scanner error is relatively simple: if the file end is found within a token,

terminate the program. If an invalid character is found, reject it and start over with the next token. Tokens that require some parsing, such as floating-point number tokens, will require attention to syntax errors whose nature should be clear from the state reached during the scan. (The scanner is a finite-state automaton.)

An unclosed comment or quoted string cannot usually be detected by the scanner; its effect must be detected by the parser.

For example, consider the following program fragment:

```
MOVE 'SAM'S FRIEND' TO BUFFER;
MOVE 'FREDDY AND MARY' TO BUFFER;
```

The quote mark just following SAM in the first line causes the two lines to be divided into tokens as follows:

```
token:  MOVE
string: SAM
tokens:  S FRIEND
string: TO BUFFER; MOVE
tokens:  FREDDY AND MARY
string: TO BUFFER;
  (etc.)
```

The remaining program is inverted—quoted strings will be regarded as program material and the program material will be regarded as quoted strings. The scanner will in general be unaware of this until an end of file or end of record is detected inside a quoted string. However, the parser will be reporting a large number of syntax errors.

A macro error usually cannot be detected until the compiler attempts to parse it. Then a simple macro error is often magnified into a voluminous error report.

A *syntax error* is detected by the parser in a READ operation. Some token x, called the *error token*, is incompatible with the state of the parser at the point at which x is to be scanned. In an LR(1) parser, x is detected in a read state P such that no transition from P on token x exists. In an LL(1) parser, x is detected through a mismatch between x and a terminal stack top token, or through failure to accept x on a nonterminal stack top in the transition table.

Error recovery in line-oriented source can be very easy. Line-oriented source is subdivided into lines, such that the parser may (for the most part) start over from a known state on each line. Thus when a syntax error is detected in some line, recovery is simply a matter of skipping the rest of the line and starting over on the following line. Whether the source is line-oriented or not depends on the language. Fortran, Basic and APL are examples of line-oriented source. Algol, Pascal, and PL/I are not.

Line-oriented recovery doesn't always work properly. For example, in Fortran, a line can contain several declarations. If some of these are skipped because of a syntax error earlier in the line, a number of semantic errors will likely be reported later in the program. Also, if a syntax error occurs in the first statement of a subroutine, special attention must be paid to getting the compiler into a state that will accept the subroutine statements.

Error recovery in free-form source, for example Algol or Pascal, is more difficult. There are no distinctive line boundaries to aid recovery. However, certain tokens such as semicolon or END could be used in a recovery. PL/I in particular has the rule that every statement is terminated by a semicolon, and the semicolon has no other purpose. Hence PL/I error recovery can be achieved by skipping source through the next semicolon.

A better strategy for free-form languages is to attempt to patch the region of the error, by dropping and/or inserting tokens such that the sentence is brought into the language. We shall discuss such approaches in section 12.3.

A *semantic error* is any error symptom in a section of source that is free of syntax errors. A semantic error is detected through some special test called out in an APPLY action. Since the parse has been successful, the nature of the error is usually very clear, and a specific diagnostic message can be easily formulated. Section 12.2 contains more details.

Semantic errors arise out of a failure to describe a programming language through its grammar alone. (However, some languages, notably APL and Algol 68, are fully defined through their grammars.) For example, the attributes of a declaration may have to satisfy some relation that would be difficult or impossible to define in a context-free grammar, or the sizes of certain literals may be restricted in some fashion.

Effect of an Error on the Compiler

Suppose that an error is detected in a source program. What changes in the compiler should take place? The answer to this question depends on the kinds of errors that are to be subsequently detected, and on the nature of the code generation process.

A block-structured, free-form program containing an error cannot usually be repaired by the compiler to the complete satisfaction of the programmer. Any repair made by the compiler is for the purpose of continuing to check for errors, not to create a "correct" program. Hence, once an error is detected, there is no point in carrying out any compiler operations beyond those needed for further error detection.

One question in error recovery design is whether semantic checking should continue after a syntax error. One important semantic check is for undeclared and misused identifiers. Should these be suppressed upon the first error or not? There are two schools of thought. If semantic checking is discontinued, then semantic errors past a syntax error are unreported. If semantic checking

is continued, it often happens that the recovery from a syntax error will cause the appearance of spurious semantic errors. Syntax error recovery in a declaration is especially likely to cause spurious semantic error messages.

It is certainly much easier to design the compiler such that all code generation and semantics checking stop upon the first syntax error. The compiler will also run faster in its "syntax-checking" mode, being relieved of an appreciable number of synthesis operations. Whether this will be appreciated by a user is not clear.

Most commercial compilers provide a limited amount of semantic checking, at least the checking that normally would be made in the pass in which an error was found. Users of compilers seem to accept the possibility of spurious error messages, particularly when the compiler reports the sort of changes it made in the user program during error recovery. Serious errors found in one pass of a multipass compiler are usually sufficient reason to end the compilation on that pass. However, all the syntax errors are found on the first pass. Since semantic errors are usually easy to repair, it is possible to continue with all the passes, given no syntax errors in the first pass.

Some compilers are designed to make incremental adjustments to a code file through *partial compilation*. For example, one procedure out of a set of procedures may be selected for a recompilation. This approach to compilation reduces compilation time during program development and facilitates writing procedures in different languages that are intended for the same program. If this is the case, the compiler is given a code file that it may update, but should not destroy. It is then important that, upon an error, the code file be closed without destroying its validity. Otherwise, a complete compilation to reconstruct the damaged code file would be required after each syntax error recovery.

A more difficult requirement for a compiler can be expressed as follows: Accept a code file and update it with the code for every procedure that can be correctly compiled, but do not update it for procedures that contain errors. Although this appears to be an impossible requirement, in fact it can be achieved. Essentially, a decision is made at the end of a procedure whether to update the code file or not. An error found in the procedure, or in a covering procedure, is sufficient to not update the file. It is still possible to damage the code file on certain kinds of syntax error, but our experience indicates that, given a reasonable error recovery strategy, the file is rarely damaged. Clearly, if any procedure code has been written to the code file prior to detection of an error, there must be some means of marking it invalid, or removing it.

12.2. Semantic Errors

A *semantic error* is any error detected in an APPLY operation on source that is free of scanner or syntax errors. Semantic tests are usually necessary,

owing to the inadequacy of a context-free grammar to fully define the language. Some examples of semantic tests required in Algol are:

- Consistency of an identifier's attributes with its appearance in the source. A statement label should not be used as an expression variable, a simple vabiable should not appear with an index, a non-typed procedure should not appear as a function, etc.
- Matched appearances of identifiers. In a procedure declaration, the formal parameters are first listed, then declared. Each declaration must correspond to one of the names in the list and vice versa.
- Limitations imposed by an implementation on the sizes of arrays, or the number of variables. These depend on the magnitude of various constants.
- Overflow in numeric arithmetic performed by the compiler.

Semantic errors are very easy to diagnose accurately, since the nature of the problem is apparent from the apply action. Recovery requires some attention to the possible side-effects. In general, some simple action should be chosen that removes the semantic problem with no undesirable side-effects.

An undefined identifier should be reported and then defined by entering it in the symbol table. Although its attributes are not always clear from its context, there is usually sufficient information that some broad default attributes can be chosen. If necessary, that symbol table entry can simply be marked "entered through error." Then any subsequent misuse of the identifier will not be reported.

Identifier misuse occurs through the appearance of some declared identifier in a context that requires a different kind of declaration. For example, a statement label in Algol 60 cannot be used as a data item; it does not stand for a numeric value. When a misuse is detected, it is usually because of a multiple use of the same identifier. It is best, we feel, to add the attribute "error" to the identifier and to suppress further complaints centered on it.

A literal value found to be out of range can easily be adjusted to a default compatible with its context.

The possibility of overflow during compilation must be considered in any arithmetic operation involving source data. In order to frame a suitable diagnostic message and to recover gracefully, each such operation in the compiler must be identified and suitably protected. Overflow can occur through the conversion of number representations to internal values, through constant folding, or through memory allocation that exceeds the bounds of the internal integer representations. These occurrences clearly call for different kinds of diagnostic messages and recovery strategies.

12.3. Syntax Errors

A *syntax error* is detected by the parser upon encountering some token that is incompatible with its current state and stack. We shall discuss error recovery for an LR parser in some detail, although an error recovery strategy can be developed for any of the deterministic parsers discussed in chapters 4, 5, and 6.

Most error recovery systems assume that one of three kinds of error exist: insertion, deletion, or substitution. An *insertion* error results from the presence of a token in the source string that, if removed, would result in a valid sentence. An insertion error is defined by the following valid and invalid derivations:

$$\sim(S \rightarrow^+ xty) \text{ but } (S \rightarrow^+ xy)$$

Here t represents the incorrectly inserted token. A *deletion* error results from the inadvertent deletion of a token that is needed for syntactic validity:

$$\sim(S \rightarrow^+ xy) \text{ but } \dagger S \rightarrow^+ xty)$$

Here t is needed to transform xy into a sentence. A *substitution* error is such that a change of one token into another will yield a valid sentence:

$$\sim(S \rightarrow^+ xty) \text{ but } (S \rightarrow^+ xuy)$$

Here a change of token t to u yields a sentence xuy.

If an error is one of these three kinds, we say that the error is *simple*. If, in addition, each pair of errors is separated by a fairly large number of tokens, we say that the errors are *sparse*. Most errors found in practice are simple and sparse.

Even if errors are sparse and simple, a parser cannot always determine which of the three kinds of error produced the error symptom. A substitution or insertion error usually results in an error symptom at the error token; however, the error symptom may also appear later. For example, a BEGIN inserted between two statements would be interpreted as a new block opening (assuming that a semicolon may be either a separator or a terminator of statements). The error symptom of the inserted BEGIN will not appear until much later in the program, when the absent END manifests itself. A deleted token can at best result in an error symptom on the token following the deletion; again, it is possible that several additional tokens are scanned before the error symptom is seen.

We see that an attempt to uncover the "real" error is usually impossible. At best, given full knowledge of the remaining input list, an error recovery strategy can indicate the least number of repairs to the source text that suffice to bring the source into syntax.

12.3.1. General Methods

Suppose that a program is known to have at most one error somewhere, of one of the three kinds above. Then the error can be identified in $O(n^2)$ operations, where n is the number of tokens in the program. We simply try each of the three error types in every possible token position, and parse the resulting string. Given one out of n sentence tokens, there is one insertion error, m replacement errors, and m deletion errors possible, where the language contains m terminal tokens. An LR parse requires Kn operations, for some fixed K, hence a single error repair (but not necessarily "the" error repair) can be identified in $K(1+m^2)n^2$ operations.

If an unknown, but bounded, number of errors is known to exist, then the previous strategy can be repeated with combinations of 2, 3, 4, ... , k errors, where k is the upper bound on the number of errors present. A minimal repair set can then be identified in $K((1+m^2)n^2)^k$ operations, or roughly $O(n^{2k})$ operations.

A strategy for unknown k might then be as follows. Let $k=1$, and explore single errors. If a single error repair is found that yields a sentence, then halt. Otherwise, set $k:=k+1$, and search for k error repairs that yield a sentence. Continue to increment k and search for a k-error repair on each failure. A valid sentence must be found before $k=n$; however, if it must continue that far, we have performed $O(n^{2n})$ operations.

We clearly need a more efficient means of repairing errors. The bound $O(n^{2n})$ is much too large for any program of practical size.

Aho and Peterson (Aho [1972c]) describe an algorithm that will parse any input string to completion in a time proportional to the cube of the input length, provided the string contains only insertion, replacement, or deletion errors. Essentially, they show how error productions can be added to a grammar G to yield a grammar G'that accepts any string in G's alphabet. A sentence is then parsed through Earley's algorithm (Earley [1970]); this algorithm will parse sentences in any context-free grammar in $O(n^3)$ operations. The parse may then be interpreted such that a minimum number of error productions are used in the parse.

Aho and Peterson have therefore shown that minimal error recovery can be achieved in $O(n^3)$ operations. For large n, this is still much too large for practical purposes. Their approach also requires a separate error parser, the Earley parser. The Earley parser may also be used for the error-free parse, but it is less efficient than an LR(1) parser in time and space.

12.3.2. Diagnosis of a Syntax Error

How meaningful can a syntax error diagnosis be to a programmer? Consider an LR(1) parser at an error symptom. The stack contains information about the previous history of the program, and the top-of-stack

state is a read state that expects a certain set of terminal tokens next; the error token is not among them.

The easiest diagnostic message to generate is a list of those tokens that are next expected; if the error token were one of them, then the error would not have occurred. For example, the message

```
   ERROR IN LINE 19.000, COLUMN 13
   A := WAS FOUND, BUT POSSIBILI-
TIES ARE: + - * / ↑ . . .
```

is a reasonable syntax error message. The list of possibilities is easily generated from the outsymbols of the read state in which the error was detected.

The top-of-stack read state is of course associated with some configuration of items as a result of the LR(1) parser construction (see chapter 6). It is conceivable that the configurations could be studied with the objective of generating an error message catalog tailored to the kinds of productions being worked on. Unfortunately, this can be an extremely laborious task—a typical grammer may yield over a thousand read states, each of which would have to be manually studied by someone. Any change in the grammar would render parts of the catalog useless, or at least require a reexamination of the messages and configurations.

Another diagnostic strategy is to simply report what was done to correct the error. The programmer is then often impressed by what the compiler is attempting to do for him and may well overlook the fact that the compiler is unable to provide a meaningful correction to the error symptom. If several tokens must be dropped or some tokens must be inserted, such a report would be useful to most users.

12.3.3. Patching a Syntax Error

At the point of an error symptom in a bottom-up LR parser, we have a stack of states. . . , S_2, S_1, S_0, with S_0 on top, and a remaining input string xa_1a_2 . . . , where x is called the *error token*. The error token x and the top-of-stack state S_0 are such that S_0 is a READ or LOOKAHEAD state, and x is incompatible with that state. State S_0 can only be a READ state if the LOOKAHEAD tables are incomplete; see chapter 6 for details.

The error recovery system should arrive at a new state stack and input string such that parsing may continue. The new state stack need not be related to the old one at all; however, the new input string should be derived from the old one by only changing the first n tokens, with n reasonably small. We call such a modification to the state stack and input string a *patch*. A patch therefore consists of some combination of the following operations:

1. Deletion of one or more left-most tokens of the input string.
2. Insertion of one or more tokens as prefixes of the input string.
3. Removal of states in the stack.
4. Pushing one or more states onto the stack.

Now the state stack can only be modified into some stack that represents a viable prefix in the language; failure to ensure this condition will result in a catastrophic failure later in the parse. The easiest way of ensuring it is to only pop states from the stack or to add a state to the stack top that is compatible with the existing stack top. A new state Q to be pushed on the stack must be such that a transition from the top-of-stack state P to Q exists in the complete LR finite-state control.

The state stack represents a condensed history of all of the source material that has previously been parsed. It carries such information as the nesting in a parenthesized structure, whether one or more procedures have been entered, whether a declaration or an executable statement is being developed, etc. We contend that such information is useful in error recovery. For example, if the stack indicates that four BEGIN's have been seen so far, the chances are good that the four matching END's will appear past an error symptom.

Scanner Feedback

As much as possible of the remaining input string should be retained in order to provide some measure of syntax checking of the rest of the program. That is the whole point of error recovery. If the recovery strategy is to reject the rest of the string and to replace it with a string compatible with the state stack, then we really don't need error recovery.

Now an LR(1) parser will require the scanning of at most one token past a handle. A reasonable strategy that we shall develop in the next section is to make use of this next token and the stack contents to devise a patch. As an improvement, we may read several tokens past the error token, in an attempt to arrive at a more effective patch. However, the use of more than one token can introduce some difficulties if the scanner is dependent in any way upon the result of any parser apply actions.

For example, the scanner may be expected to classify identifiers by some criterion in order to return one of two or more token codes associated with identifiers. The scanner might have several states that guide its lexical decisions, the states being partially set by apply operations. If such is the case, we say that *feedback* exists between the apply operations and the scanner. It is very desirable that no feedback from the apply actions to the scanner exist.

If feedback exists, then the scanning of tokens past the error token is not reliable. There will be no apply operations resulting from such a scan to guide the scanning, and the token codes supplied by the scanner are likely to be incorrect.

Feedback should be avoided in the design of a compiler, regardless of the error recovery strategy or of the apparent need for it in the language. If necessary, the scanner may be based on some automaton whose state transitions are determined by the tokens scanned, in order to identify certain special tokens without feedback.

Given a feedback-free scanner, it is relatively easy to scan several tokens in the input string past the error token. These tokens may then be used either to construct a patch that is compatible with these tokens or to discard more input tokens until a patch can be devised.

Terminal or State Insertion

Consider a choice between inserting a terminal token T or pushing a state P. State P corresponds to some terminal or nonterminal token T′. If T′ is terminal, then pushing P is clearly equivalent to inserting T′. Suppose that T′ is nonterminal, and that pushing state P causes the parser to accept the next input token, and of course that P is compatible with the stack-top state. We then effectively have a sentential form

$$wST'xy$$

where wS represents the stack, with S the stack-top symbol, T′ is the inserted nonterminal, x the error token, and y the remaining input list. Now the next derivation step must be of the form

$$wST'xy \Rightarrow wSt'xy$$

where a production $T' \rightarrow t'$ exists. If $|t'| = 1$, we again see that an insertion of t′ is equivalent to pushing P; by inserting t′, the LR(1) parser will reduce t′ to T′, since t′must be the handle of wSt′xy.

We conclude from this discussion that if a nonterminal T′ can derive a single terminal or nonterminal symbol, then any state whose insymbol is T′ need never be pushed on the stack during error recovery; it is sufficient to insert one of the derived terminals. We need only consider pushing those states P whose insymbol T′ is such that T′ derives either the empty string or strings of length greater than 1.

A similar result holds for the insertion of terminal strings of length 1, with $1 > 1$. However, the number of possible string insertions grows exponentially with the string length, and we feel that it is infeasible to consider the insertion of more than one terminal token.

Now most terminal token data structures are empty, while many of the nonterminal data structures are nonempty. Further, a simple default data structure can always be selected for those terminals with a structure, while the nonterminal data structures are often more complicated, perhaps requiring links to the symbol table or to other structures. It is clearly desirable to reduce the number of inserted nonterminals to a minimum. An inspection of the

grammar is sufficient to find those nonterminals for which no terminal insertion is an equivalent substitute. It is then feasible to add special error productions for each of these nonterminals, of the form $T \rightarrow x$, where $|x| = 1$ and x can be any terminal. The appearance of such a production can only occur during error recovery, but can be exploited by the apply actions to create a suitable data structure for T.

Given a grammar such that for every nonterminal T, a production of the form $T \rightarrow x$ exists, with $|x| = 1$, then nonterminal insertion as an error recovery strategy is unnecessary. Only terminal insertion, state dropping, or input token deletion is needed.

12.3.4. Semantics Operations in Error Recovery

Recall that an abstract-syntax tree (AST) is built as a linked list in a semantics stack, and that the semantics stack is associated with the parser's state stack (see chapter 7.) Now assume that some semantics operations are to be continued through a syntax error recovery. It is obviously vital that the integrity of the semantics stack structures be preserved through any error recovery patch.

If states can be removed in a patch, then no forward links in the state stack data structures should exist. That is, there should never be a pointer from some structure in the stack to a structure closer to the stack top. All pointers, if any, must be directed from the stack top inward.

If a state is pushed on the stack through a patch, then a data structure compatible with that state should be pushed as well. The structure will correspond in general to some terminal or nonterminal symbol in the grammar associated with the pushed state. If several kinds of structure are possible, the least complicated should be chosen. If necessary, special "null" structures might be devised in order to avoid difficulties in creating a suitable data structure.

We see that every new pushed state should be associated with some data structure valid for that state, and that pointers in a data structure should never point to structures associated with states higher in the stack. If these rules are rigidly adhered to, then the synthesis operations may be continued through an error patch to whatever level seems desirable for effective semantic error reporting.

12.3.5. A Bounded-Range Error Recovery Strategy

We first develop an error recovery system that does not require a lookahead of more than one token in the input string. It is therefore suitable for a compiler with feedback to the scanner.

1. *Upon encountering a syntax error*, the stack will consist of the states

$$S_n, S_{n-1}, \ldots, S_1, S_0$$

where S_0 is the stack top state. Let X be the error token (next in the input list) and let S_0 be a READ state. Then X is not among the tokens in the transitions from S_0. Set a flag ERFLAG that indicates that error recovery is in progress. Make a copy of the stack for the sake of stack restoration. Set variable $j := 0$; j will mark the current stack top. In the following steps, the operation "restore the stack" will result in the stack

$$S_n, S_{n-1}, \ldots, S_{j+1}, S_j$$

This is the original stack, less the top j states.

2. *Terminal insertion.* For every terminal T that is accepted by state S_j, insert T before X in the input string and attempt to parse the input string through token X (but not past X). If a T exists such that the parse is successful, we have found a patch. Otherwise, restore the stack (after each trial) and continue.

3. *Nonterminal insertion.* Let S′ be some state that is compatible with S_j. (There exists a transition from S_j to S′ on a nonterminal in the complete LR FSA.) Push S′ and attempt to parse X with the resulting stack. If an S′ exists such that the parse is successful, we have found a patch. Otherwise, restore the stack and continue.

4. *Dropping a state.* If $j = n$, go to step 5. Otherwise, set $j := j - 1$. Attempt to parse through X with the new stack. If successful, we have a patch. Otherwise, restore the stack and go to step 2.

5. *Bottom of stack.* Here $j = n$. Set $j = 0$. Restore the stack. Read the next token, call it X, then go to step 2.

Upon finding a patch in the above algorithm, the stack should be restored and ERFLAG reset. We know that the stack will represent a viable prefix, and that a parse through the current next token X must succeed. Whether the parse will succeed through subsequent tokens is unknown.

The nature of the patch depends on the order in which insertion tokens are produced in steps 2 and 3. It may happen that several alternative patches are possible with a given j and X; the algorithm simply accepts the first one that it finds.

The nonterminal insertion operation 3 may be eliminated through the use of error productions, as explained in section 12.3.3. Elimination of this step eliminates considerable difficulty with the creation of a suitable semantics stack structure associated with the nonterminal or, more accurately, places their creation in the apply operations where they belong.

Discussion

We have found that the recovery with this simple strategy is not particularly effective. There are several problems as follows:

- There is a tendency to remove more states from the stack than is really necessary. Spurious error reports are often the result of removing too many states from the stack.
- Often an inserted token is an unfortunate choice. For example, we found that a relatively common syntax error caused the token CASE to be inserted under this algorithm. The CASE statement is highly structured, and the insertion of CASE is rarely needed, hence many subsequent spurious errors were generated.
- The information in the input list past the error token X is not used. This information could be used to more effectively select j or an insertion token or to force more input scanning.

12.3.6. Variations on the Bounded-Range Strategy

Several variations on the bounded-range strategy can be devised to correct some of the deficiencies given above. Of these, we have studied error recovery based on the first two rather extensively. A discussion of the strategy's effectiveness is given in section 12.3.9.

1. The tokens that may be inserted may be limited to a subset of the grammar's alphabet. In addition, a limit may be placed on the degree to which the stack can be reduced by dropping states. We find the following guideliness to be effective.

(a) Select a set of tokens that should not be inserted. We feel that these should be terminal tokens and that each one would, if it were to be inserted, demand some subsequent structured form that is unlikely to be present in the source. Candidates for the "don't insert" list are such keywords as PROCEDURE, IF, CASE, WHILE, VAR, to draw from Pascal as an example language.

(b) Select a set of states S′ such that if $S' = S_j$, then j will not be decremented. State S′ can best be chosen by selecting a set of tokens T′ such that the states S′ are accessed by the members of T′. We feel that the members of T′ should be those terminals and nonterminals that represent the heads of major structures in the language. Such tokens as PROCEDURE, CASE, BEGIN, etc. are reasonable candidates for this list.

Once these two lists have been chosen, the error recovery strategy given above is modified by permitting an insertion only of tokens not in the list (a) and not reducing the stack below a state in the set S′, list (b).

These lists provide a means of tuning the recovery algorithm for a particular language and its grammar. Effectively, they convey information to the recovery algorithm regarding the kinds of errors that are expected in practice. It is our opinion that certain kinds of error are difficult to correct satisfactorily by whatever means. These usually stem from failure to note critical keywords that introduce major program structures. If one of these keywords is misspelled or absent, it is hard to imagine an error recovery strategy that can infer that problem without a major analysis of the remaining input list, especially if some other defective patch is compatible with the information at hand (stack and following few tokens).

Given such a difficulty with critical keywords, it is reasonable to conclude that, if present, they are important recovery indicators, not to be discarded lightly, and if absent, they should not be inserted.

2. As an improvement on variation (1), report a syntax or semantic error only if no error has been reported on the last m tokens, where m is some small number. That is, if an error is seen on a token a_0 in the string $a_0a_1...a_k...$, then we report an error found at token a_k only if $k>m$. Note that only error reports are affected, not the recovery. This variation results in an improvement, however, since the chances are good that an error found in the wake of a previous error is spurious and caused by poor recovery rather than a genuine error. Since each error resets the token count, several sequential error reports may be suppressed.

Of course, this improvement increases the risk that a real error will go unreported.

3. If the compiler is feedback-free, accept a patch with the above error recovery strategy only if the next m tokens in the input list are accepted, where $m>1$. By attempting to scan more than one token in the input list, the chances of finding a valid recovery are increased. Essentially, there are more clues if more tokens are considered; the possible patches are more constrained. Of course, if no patch can be found, input tokens must be dropped. Eventually, either the source is completely scanned or a patch is found.

This variation may be combined with (1) and (2) above, as an improvement.

12.3.7. Forward Move

Given that the objective of error recovery is to continue parsing with a minimum of subsequent spurious error reports and with a maximum of checking of the subsequent input text, it would appear that the input text past the error point is of considerable importance to an effective error recovery strategy.

We owe the notion of a *forward move* to Graham and Rhodes (Graham

[1975]), who studied error recovery in an operator precedence parser. A forward move is essentially an attempt to reduce, through parsing actions, a portion of the text following the error token before selecting a recovery strategy. Those reductions are made that would have to be made in any parse. A forward move reduces the number of trials needed for a feasible error recovery solution and increases the forward context information needed for an effective recovery.

With operator or simple precedence, a forward move is relatively simple. The error token and the remaining input list is assumed to be some string to be parsed, with the usual "start" token included as a prefix. Reductions are then made as usual, halting when another error is found. Recall that an error is detected through either a token pair that is not in the precedence table, or through a handle that fails to match any production right member. The forward move may also continue to the end of the input list.

Penello [1978] has developed a forward move algorithm suitable for use with an LR(1) parser. In his scheme, an LR parse is begun just past the error token (it could also just as well begin with the error token), with a special parser called an *error parser*. Each state of the error parser represents a set of states of the LR(1) parser. The initial state consists of all those read states that can accept the first input token of the remaining input list. The error parse continues through read, lookahead, and apply actions, just as in an LR(1) parser, except that every action must agree (in a sense to be described more precisely next) and any apply action must be such that there are sufficient states on the stack to support the action.

Let Q′ represent a set of states in the error parser and T some token to be next read. Then a transition in the error parser on (Q′,T) is permitted only if each of the states q′ in Q′ either (1) have no move on T or (2) those that show a move agree in action. That is, if any of the Q′ states call for a READ action, then every state must either call for READ or for ERROR. (An ERROR simply means that some state will be dropped from the state set on the move.) If any of the Q′ states call for a APPLY, then every state must either call for a APPLY on the same production or for ERROR. If any of the Q′ states are ACCEPT, the forward move halts. Finally, if an APPLY move is indicated, and there are insufficient states in the forward move stack to support the apply action, the forward move halts.

A special stack is used for the forward move, initially containing only the state set accepting the first token. Given a READ action, a set of states representing the successor states of the members of Q′ on token T is pushed on this stack. Given an APPLY action such that sufficient states exist on the stack to support the action, those states are popped and replaced by the indicated apply set.

The action of the forward move is essentially to perform a number of reductions of the input following the error token. It condenses the information in the input list and can be used to devise a patch. It is possible that the

forward move may continue to the end of the program and perform a large number of reductions. It may also halt within one or two tokens, either due to another syntax error, or for some other reason.

Consider a right-most derivation

$$S \rightarrow^* vAy \rightarrow vwxy$$

where $A \rightarrow wx$ is a production. vw is some viable prefix of the sentential form vwxy. We say that z is a *viable fragment* of the viable prefix if z is a suffix of vw. Further, we say that U is a *derived viable fragment* of the sentence suffix zy if (1) U derives z, and (2) during a parse of any sentence ending in zy, at some point the parser must reduce z to the viable fragment U.

Penello shows that his forward move yields a viable fragment U that represents the largest possible subsequent input string. Let that string be z. Then $U \rightarrow^+ z$ is a derivation that must be made eventually by any parse.

12.3.8. Correction Strategies with a Forward Move

Pennello proposes a heuristic for error correction, given an existing state stack and a forward move. It is a variation on the bounded-range error correction strategy given in section 12.3.5. However, he can make effective use of the forward move information by looking for a correction that incorporates the states beyond the error point. This amounts to requiring that any repaired configuration be parsable far beyond the error token. An experimental evaluation of his scheme is given in 12.3.9.

Mickunas and Modry (Mickunas [1978]) propose reversing the parse represented by the state stack in some cases in order to achieve an effective patch. This involves much more than just dropping states—it must be possible to reconstruct the parse steps in reverse and obtain some previous stack. A parse reversal can only be achieved by maintaining a complete history of the parse, or by keeping on hand the tokenized input string, along with input string positions in the stack. This information must be kept for any parse, and represents a cost of compiling error-free as well as erroneous source text. Parse reversal also carries significant semantic implications—the semantic actions should also be reversed if the intention is to continue semantic actions through an error recovery.

However, if a parse reversal is possible, their approach apparently yields a good repair of a number of otherwise difficult syntax errors.

Their algorithm consists of two phases—a *condensation phase* and a *correction phase*. The condensation phase is essentially a forward move. A set of states S that can shift on the error token is determined, and a parse is made of the remaining input string, using one of the states in S as a start state. A forward move terminates in one of two ways: (1) an attempt is made to reduce

over the error point, or (2) another error occurs. In case 2, the parse configuration is called a *holding candidate*, and is held for possible use when other strategies fail. In case 1, the configuration is called a *correction candidate*.

At least one correction candidate will always be found– it may be the error configuration itself. If several are found, they are independently examined.

The *correction phase* is a systematic recovery strategy that incorporates the following unit operations:

- Insertion of a terminal token.
- Retreating the parse. By this, they mean reversing the parse represented by the state stack to some former configuration
- Dropping a state representing a terminal token from the stack. If no terminal token is on the stack top, then a parse must be reversed until one is. When a state is dropped, the set of possible insertion tokens changes.
- Assigning a cost to each insertion and each stack deletion, and accumulating a total cost for some strategy path.
- Abandoning a strategy path when its cost exceeds a preset limit and launching an alternative path.
- If all else fails, a holding candidate is selected, and the entire condensation-correction strategy is recursively invoked on it. Recall that a holding candidate is generated when a second parsing error is encountered during a forward move.

There are usually many different paths to follow, depending on the results of the condensation phase and on the number of insertions that are found to be compatible with some stack and the condensation states. A path is followed until either a successful repair is found, or the net cost exceeds a preset threshold. All the paths are followed, and a repair with least cost (if any) is selected.

The correction phase will never drop the error token. If it succeeds in finding a path involving one or more of the operations listed above it reports success, and an effective recovery patch is found. If it fails, then the error token must be dropped and new condensation and correction phases must be launched.

This error recovery scheme was implemented on a small grammar containing about 40 productions and 356 states. Some examples of recovery follow.

Example 1.

The program segment

```
. . . READ A B[20] WRITE A; GOTO . . .
```

should carry a semicolon just before the WRITE. The error configuration is

```
. . . READ <input-list> <identifier> [ <expr> ]
          ? WRITE A; GOTO . . .
```

where "?" marks the error. The condensation phase produces one correction candidate:

```
. . . READ <input-list> <identifier> [ <expr> ]
          ? <statement> ; GOTO . . .
```

associated with two possible reductions:

```
<statement> ::=G= <identifier> : <statement>
```

and

```
<statement-list> ::= <statement-list> <statement> ;
```

In the first case, no single insertion repair is possible, so the parse must be reversed. Token "]" is dropped, <expr> is then expanded to "20", the "20" is dropped, the "[" is dropped, and this path terminates due to excessive cost. (Each insertion or deletion costs 2 units. A path is terminated when the net cost exceeds 5 units.)

This leads to the second case. Here the insertion ";" provides a repair at a cost of 2.

Example 2.

The program segment

```
. . . X := Y : A := B ; GOTO . . .
```

is repaired if the colon after Y is replaced with a semicolon or ":=". The error configuration is

```
. . . <statement-list> <leftpart> <identifier> :
        ? A : = B ; GOTO . . .
```

In order to see why this configuration is reached, a <leftpart> production is

```
<leftpart> ::= <identifier> : =
```

The characters ":" and "=" are considered separable for parsing purposes, hence a missing "=" component of the expected ":=" has produced the given error configuration.

The condensation phase produces two correction candidates:

. . . <statement-list> <leftpart> <identifier> :
? <assignment> ; GOTO . . .

and

. . . <statement-list> <leftpart> <identifier> :
? <statement> ; GOTO . . .

The first of these stems from an attempted reduction

<assignment> ::= <leftpart> <assignment>

and the second from two possible attempted reductions:

<statement> ::= IF <boolean-expr> THEN
<statement> ELSE <statement>

and

<statement> ::= <identifier> : <statement>

In the first of these, only the <statement> following the ELSE can possibly be associated with the <statement> following the "?" in the correction candidate. Recall that a forward move disregards the state stack preceding the error point, hence has no basis for disregarding the obviously complicated attempted reduction.

We therefore have three cases, one from the first correction candidate and two from the second.

In the first case, the correction phase finds an insertion repairing the error at a cost of 2 units.

In the second case, the correction phase quickly exceeds the threshold cost with stack deletions. It effectively attempts to find an IF-THEN-ELSE statement in the stack and its reversed parsing. Since no such statement exists, this path can only drop states from the stack until the cost threshold is exceeded.

In the third case, the correction phase backs over ":", backs over <identifier>, expands <leftpart> to "<identifier> :=" then deletes "=" (at a cost of 2), to yield the repaired configuration

. . . <statement-list> <identifier> : <identifier> :
<statement> ; GOTO . . .

Example 3

The program segment

. . . BEGIN X := Y ; Y := Z ; WRITE X Y ; .

requires either dropping the BEGIN or inserting "END ;" before the terminating period. The reduction of this segment eventually yields the error configuration

. . . <statement-list> BEGIN <statement-list>
<statement> ; ? .

The condensation phase has only the period to work on; however this token appears as part of the production
<program> ::= <declaration-list> <statement-list> .
It therefore selects this reduction, and yields a correction candidate identical to the error configuration. The correction phase then fails to find a token insertion (the grammar requires insertion of "END ;", not just "END".) It therefore backs up over ";", <statement>, <statement-list>, and deletes BEGIN, at a cost of 2. (Backups and reverse parses carry no cost.) This yields a repaired configuration

. . . <statement-list> <statement-list> <statement> .

The repaired configuration is not right-most canonical; the <statement-list> productions are:

<statement-list> ::= <statement-list> ;
::= <statement-list> <statement> ;

Discussion

Mickunas' scheme appears to yield a very effective recovery in a number of difficult error situations. However, it requires not only a forward move, but a reversal of the parse in the stack. In addition, the repaired configuration will not necessarily represent a right-most canonical sentential form. (See example 3 above.) It is also not clear at what point a path should be terminated. If the termination cost is set too high, then the number of cases to examine expands voluminously; if set too low, the scheme will drop tokens unnecessarily.

As we have seen, a forward move requires a pure syntax—one in which the recognition of every token can be achieved by the scanner alone with additional semantic information such as might be contained in a symbol table. A parse reversal has more serious semantic implications. It may be simplest to just abandon all semantic actions on the first syntax error, since the needed repairs to the semantics stack are so difficult. However, it then becomes necessary to abandon all hope of checking for semantic errors after the recovery is complete.

12.3.9. Empirical Study of Error Recovery

A realistic view of the error recovery problem is that an optimal solution with a reasonable time bound is unlikely to be discovered in the near future. Error recovery is now, and may continue to be, achieved by heuristic methods, with less than optimal effectiveness.

Given this situation, the relative merit of two error recovery algorithms is impossible to assess on theoretical grounds alone, assuming that reasonable means are used to reduce the computational complexity of the methods.

Nevertheless, we would like a reasonably objective means of evaluating an error recovery strategy, for the sake of selecting an error recovery strategy. We propose an empirical measure of error recovery, based on the systematic analysis of a "random" sample of programs, each containing one error.

There are of course a large number of random variables to consider. The language and its grammar have an important influence on error recovery. The kinds of errors that programmers typically make is also of interest. An occasional poor error recovery is excusable if the strategy performs well on the more frequent errors. For example, the use of a semicolon as a statement terminator (i.e., just before an END or an ELSE) is a common error in Algol 60. A mispelling of PROCEDURE is less common, if for no other reason than the infrequent appearance of the keyword PROCEDURE compared to the semicolon. If a strategy recovers well from semicolon errors, but badly from a misspelled PROCEDURE, then it is likely to be acceptable for all practical purposes.

Finally, the error recovery method itself may require some tuning, for example, in a weight function, or in the selection of "don't insert" tokens, etc.

We propose evaluating error recovery through an experiment with a large number of programs. The programs need not be identical, but must contain exactly one error each, preferably near the beginning of the source. If the error recovery strategy reports exactly one error and recovers within one token, ready to detect more errors, then it has performed satisfactorily. We find instead that in many cases, two or more errors are reported because of a nonoptimal error recovery, or that a large number of tokens are rejected. We may then interpret the relative number of single error reports and the number of rejected tokens as a measure of effectiveness of the error recovery strategy.

We leave unresolved the issues of language, grammar, typical programs, and typical program errors. Research is clearly needed in these areas. What features of a language or grammar cause error recovery to be particularly difficult? Given a language, what errors are most likely to be made by programmers? We have no answers to these questions.

An Error Recovery Experiment

We now describe an experiment in error recovery in some detail, using the bounded-range strategy described in section 12.3.5, with variations 1 and 2.

In variation 2, a syntax error is reported only if no syntax error was found within the last m tokens. Results are obtained with m=0 (i.e., every error reported) and m=3 (report made only if no report in the last 3 tokens) and compared.

A single program of 37 lines, called the *basis* program, was used for the experiment. Seventy-five copies were made, and a single error was introduced into each copy.

The error-free basis program is given below. It is written in a language similar to Algol 60. This program contains some outer block declarations, a procedure, some control structures, a procedure call, and several assignment statements with expressions.

```
INTEGER I,J,K;
REAL CONSTANT ARRAY RCA:=(15,17,29,30);
REAL ARRAY R(75);
PROCEDURE SAM(X,Y);
  INTEGER X;
  REAL ARRAY Y;
BEGIN
  INTEGER L:=16,M;
  REAL ARRAY RA(18);
  L:=25*M-RA/(I+J);
  X:=INTEGER(Y(3));
  IF X<25E6 THEN
  BEGIN
    L:=25;
    X:=55
  END
 ELSE
  BEGIN
    L:=15;
    X:=50
  END;
END;
CASE I+15 OF
BEGIN
2: I:=25;
5..9: J:=55;
ELSE: IF I=5 THEN J:=65;
END;
WHILE J>0 DO I:=I+1;
SAM(J,R(3));
I:=J:=25;
SAM(I,R);
```

```
K:=55;
K:=K+(I-J);
```

The errors introduced into the 75 copies are classified as follows. This classification is based on the modification made to the original source to yield the defective source. In most cases, the error symptom could be interpreted in more than one way, however:

- 10 misspelled keywords (BEIGN for BEGIN, IFF for IF, etc.).
- 1 interchange of two keywords.
- 32 token insertions.
- 18 token deletions
- 10 token substitutions
- 2 misspelled user identifiers
- 2 misplaced statements, i.e., a declaration appearing within executable statements.

Here are two examples of error recoveries, one "good" and another "bad." These examples also illustrate a particular reporting scheme.

Example 1: A "Good" Recovery

```
          .
          .
          .
10.000     0  1   INTEGER L:=16,M;
11.000     0  1   REAL ARRAY RA(18);
12.000     6  1   L:=25*(M-RA/(I+J);

ERROR IN LINE 12.000
A ; WAS FOUND, BUT POSSIBILITIES ARE: )

STACK: <START> <DECL-LIST> ; <PROCEDURE-HEAD>
 <PROC-FP-HEAD> <BEGIN> <LOCAL-DECLARATIONS>
 <RREF> := <TERM> <MULOP> ( <EXPR>
TOKEN ) INSERTED
13.000     22  1   X:=INTEGER(Y(3));
14.000     30  1   IF X<25E6 THEN
          .
          .
          .
```

Background: From left to right in the first line is the edit line number (10.000), location of the instruction (0), a BEGIN-END level count (1), and the source line (INTEGER L:=16,M;).

Line 12.000 contains the syntax error—a missing right parenthesis. The error report given is easily generated from the LR(1) tables and the state stack. The error token is the semicolon found at the end of line 12. A number of reductions and lookahead transitions have occurred, such that only a right parenthesis is acceptable to the top-of-stack state, corresponding to the nonterminal <EXPR>.

The stack is listed, for the sake of compiler development. It would be meaningless to the average user and should be suppressed in released versions of the compiler.

The recovery selected is exactly optimal—insertion of a right parenthesis. Note that a terminal insertion with no stack reduction is attempted first; a parse through the insertion and the following token (;) was successful, and a viable solution has therefore been found.

Example 2: A "Poor" Recovery

```
                .
                .
                .
   14k200      30  1   IF X<25E6 THEN
   15.000      35  1   BEING
   16.000      35  1     L:=25;

ERROR IN LINE 16.000 OF MODULE SPLTEXT
BEING IS NOT DECLARED.

ERROR IN LINE 16.000 OF MODULE SPLTEXT
A <IDENTIFIER> WAS FOUND, BUT POSSIBILI-
TIES ARE: :=

SEE PAGE 0001, LINE 16.000 FOR PREVIOUS ERROR

STACK: <START> <DECL-LIST> ; <PROCEDURE-HEAD>
 <PROC-FP-HEAD> <BEGIN> <LOCAL-DECLARATIONS>
 <STLST> ; <IF-HEAD> <RREF>
TOKEN := INSERTED
   17.000      41  1     X:=55
   18.000       0  1   END
   19.000       0  0  ELSE

ERROR IN LINE 19.000 OF MODULE SPLTEXT
```

```
A ELSE WAS FOUND, BUT POSSIBILITIES ARE: ;

SEE PAGE 0001, LINE 16.000 FOR PREVIOUS ERROR

STACK: <START> <DECL-LIST> ; <PROC-LIST>
TOKEN ELSE DROPPED
   20.000      0  0   BEGIN
TOKEN ; INSERTED
   21.000      0  1      L:=15;

ERROR IN LINE 21.000 OF MODULE SPLTEXT
L IS NOT DECLARED.

SEE PAGE 0001, LINE 19.000 FOR PREVIOUS ERROR

   22.000      2  1      X:=50

ERROR IN LINE 22.000 OF MODULE SPLTEXT
X IS NOT DECLARED.
SEE PAGE 0001, LINE 21.000 FOR PREVIOUS ERROR

   23.000      4  1   END;
   24.000      4  0 END;

ERROR IN LINE 24.000 OF MODULE SPLTEXT
A END WAS FOUND, BUT POSSIBILITIES ARE: <IDENTI-
FIER>
 <PROC/SUBR-IDENTIFIER> . . .

SEE PAGE 0001, LINE 22.000 FOR PREVIOUS ERROR

STACK: <START> <DECL-LIST> ; <PROC-
LIST> <XSTARTX>
 <STLST> ;
TOKEN END DROPPED
TOKEN <PROC/SUBR-IDENTIFIER> INSERTED
   25.000      5  0 CASE I+15 OF
   26.000     10  0 BEGIN
   (Note:  Parsing is successful from here on)
```

Here, a misspelled BEGIN triggered three syntax error reports and three semantic error reports. The first report is a semantic report that BEING is undeclared (line 15.000). Since BEING is not recognized as a reserved word, the compiler assumed that it was an identifier.

This semantics report should have followed line 15 immediately. How-

ever, no apply action was taken on BEING until a lookahead of one token was made. This lookahead required the next source record, and the source records were printed as read. Thus line 16 was read, some reductions were made, and eventually the semantic error associated with BEING was detected.

This kind of misdirected error message can be avoided by carrying a source token position indicator in the semantics stack. Then an error associated with some token can be accurately connected to a source line and token.

This semantics report is followed immediately by a syntax error report. The stack indicates that an assignment statement, with BEING the left member, was selected as a feasible recovery solution. This solution is valid for the following line 16, "L:=25"; multiple assignments of the form

$$A:=B:=C:=25$$

are legal in this language. So far, so good.

However, a BEGIN is missing in this program, and its absence is noted in line 19, when an ELSE is found unconnected with the IF-THEN started in line 14. Note that this connection could have been detected by a forward move. The ELSE is dropped, and a semicolon is inserted to repair problems related to the dropped ELSE.

Something else has happened—the absent BEGIN has effectively caused the end of the procedure SAM in line 18. Then the local variables of SAM become undeclared, resulting in two semantic errors (lines 21, 22).

Finally, the extra END in line 24 must be accounted for; the only viable solution is to drop it, leaving a semicolon out of place. A dummy procedure call (a legal statement) is inserted before the semicolon, and the remaining program is thereby accepted without error symptoms.

Summary of Error Recovery Experiment

The performance of our error recovery strategy in the 75 programs is summarized by the bar charts of figure 12.1. Each chart shows the fraction P (ordinate) of programs that exhibited n error reports (abscissa), with n = 1, 2, 3, 4, 5, and greater than 5. Charts (a) and (b) illustrate an error recovery in which every error is reported, and charts (c) and (d) illustrate error recovery with variation 2 (error report suppressed if within 3 tokens of a previous error symptom). Charts (a) and (c) are for syntax errors only, and charts (b) and (d) are syntax and semantic errors. Every semantic error was reported, whether near a previous one or not. If nearby semantic errors were suppressed, we would expect some improvement in graph (d).

Graph (c) shows that the bounded-range strategy is quite effective, if only syntax errors are considered. Over 80% of the recoveries were optimal (one error symptom reported), and none of the recoveries reported more than 5 symptoms. If the semantic errors are also considered, the recovery is only 60% effective (graph (d)), and about 6% of the recoveries generated more than 5 symptoms.

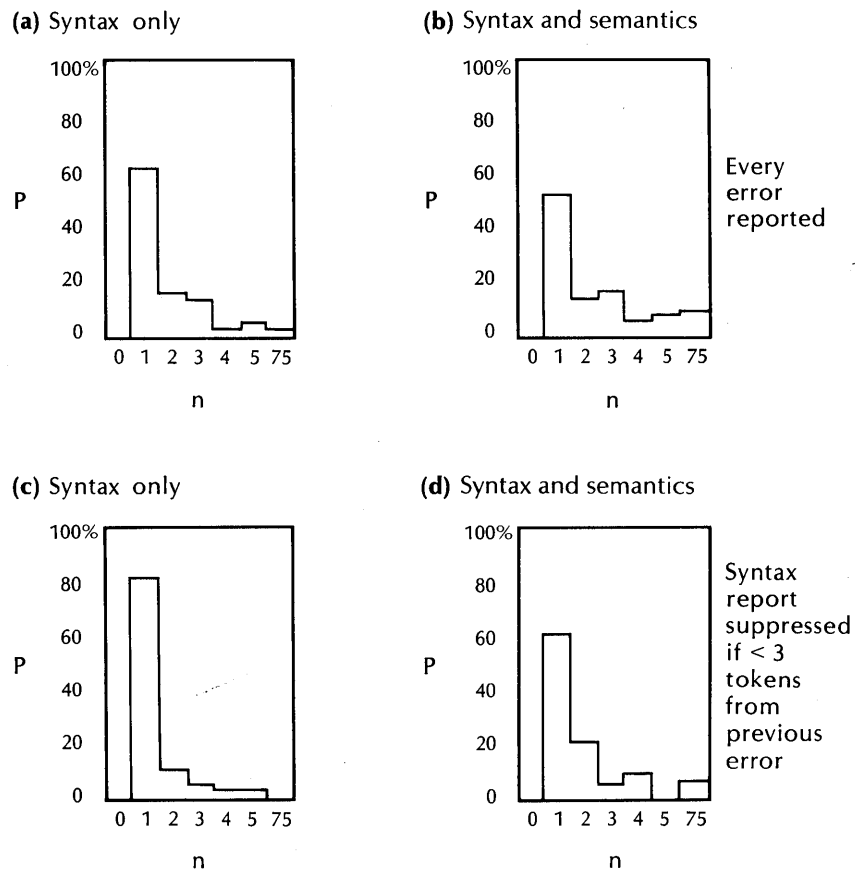


Figure 12.1. Summary of error recovery experiment. P = percentage of programs with n reports, n = number of error reports.

Very few input tokens are dropped in the average recovery, as indicated by the distribution in figure 12.2. No tokens were rejected in about 57% of the cases, and one token was dropped in 27% of the cases. In no case were more than five tokens dropped. Thus 84% of the recoveries were effected by dropping one or no tokens.

These results may be somewhat misleading, since the program chosen was rather small. The recovery is sometimes very bad in a large program. For example, an error in a long list of repeated constants or expressions sometimes causes a cyclic recovery failure in which a large number of spurious errors are reported or a large number of tokens are dropped.

Results of Pennello's Forward-Move Recovery Strategy

Pennello [1978] reports a similar experiment, on 70 Pascal programs. Each

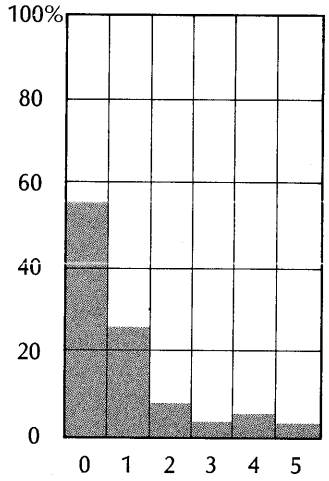


Figure 12.2. Distribution of tokens rejected during error recovery.

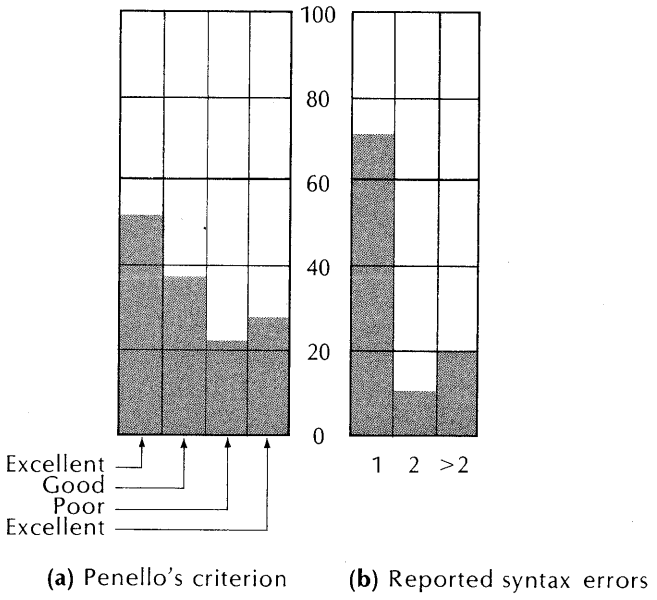


Figure 12.3. Error recovery based on Pennello's forward move algorithm—70 Pascal programs studied.

of these programs is different, but contains one syntax error. They were student programs, with student errors. His results are summarized in figure 12.3. Graph (a) represents Pennello's classification of recovery—"excellent," "good," "poor," and "unrepaired." The "excellent" and "good" categories

are in fact optimal recoveries—in every case the recovery generated just one error report. The "poor" category generated two reports for a single error. The "unrepaired" category are those in which no repair was selected, but the remainder of the program was parsed through the forward move machine rather than the parser. These rob the system of upper-level parsing, and destroy the detection of subsequent errors.

We conclude that Pennello's system is somewhere between 60 and 90 percent effective, depending on how one regards the "unrepaired" errors. However, these results are for syntax errors only. He does not state how many of the "good" error recoveries would generate spurious semantic error messages.

12.3.10. Recovery in a Recursive Descent Compiler

A recursive descent compiler is closely related to an LL(1) compiler, as we have seen in chapter 4. The stack in a recursive descent compiler is hidden in most implementations; it is the stack used to support recursive procedure calls and local variable allocations. When the stack is so hidden, through the medium of a language implementation, it is impossible to explore stack cutbacks on a trial basis. However, trial parses on the existing stack can be explored through special procedure calls.

Another constraint on error recovery exploration is the distributed nature of the parser. There are no centralized tables that can be consulted in an efficient attempt to find a patch; the parsing system is scattered through many procedures related only by their calls.

Usually, only token deletions and a one-time procedure return is utilized for error recovery. The compiler scans to some characteristic statement end token (i.e., semicolon or END), then forces enough procedure returns to place the system in a state that can accept the string following the statement end. This state is usually some procedure that can accept statements or declarations.

Although crude, this strategy is effective for languages with well-delimited statements such as Fortran, PL/I, or Basic.

Exercises

1. A CASE statement in a certain language requires that every statement in its field of statements carry at least one numeric label. The labels must non conflict. Design an effective semantic error recovery system that will not only report every possible error, but will never cause spurious errors. A sample of such a CASE statement is the following:

```
CASE <expr> OF
   15..25: 28: <stmt> ;
   13: 29: 3: <stmt> ;
   7..8: <stmt>
ENDCASE
```

The pair of periods ".." indicates an inclusive range of labels. Thus "15..17:" is equivalent to "15:16:17:". Although the labels may be in any order, it should be possible to detect missing labels and conflicts immediately.

2. Carry out an error recovery experiment similar to that given in section 12.3.9 on a commercial Algol, PL/I, or Pascal compiler. What do you conclude?

3. Rewrite the bounded-range recovery algorithm, using variation 3 and no nonterminal insertions. Design suitable data structures for the algorithm, showing that only the usual LR tables are needed, and write the algorithm in a high-level language.

4. Spelling correction is sometimes proposed as an error recovery alternative. Under what circumstances might spelling correction be a useful strategy? What language properties facilitate spelling correction? See Morgan [1970] for a popular spelling corrector.

12.4. Bibliographical Notes

The simplest recovery technique that is essentially language independent is the so-called "panic" solution. When an error is detected, input tokens are discarded until a special terminating symbol, such as ";" or "END" is seen. Then the state stack is erased until a state compatible with this end token is seen. Leinius [1970] reports a slightly more sophisticated version for an LR parser; his is the first error recovery technique reported for an LR parser. James [1972] describes an implementation of Leinius's method, with some recovery statistics.

In a top-down compiler, the predictive nature of the compiler can be used to insert one or more tokens. Irons [1963] describes such a system.

Levy [1975] and LaFrance [1971] proposed a nondeterministic recovery system that carries out a set of parses, one for each of a set of possibilities. Unfortunately, if this algorithm is carried out for more than a few steps, the resulting computation becomes unreasonably large; as we have seen, such a strategy has exponential complexity.

A minimal error correcting algorithm is given by Aho and Peterson (Aho [1972c]), described earlier.

McGruther [1972] describes a syntax error recovery and correction system that requires the grammar to be both LR and RL. (An RL grammar is such that sentences can be parsed in reverse by an LR parser.) On detecting an error, an RL parser is applied to the remaining string in reverse. The system provides a strong base from which an error analysis may proceed. The essential problem here is to locate particular key tokens past the error point from which a reverse parse may begin; it is impractical to parse from the end of a program back to the error point (that process can halt on another error deeper in the program). The selection of key symbols is discussed at some length; their selection appears to be language dependent.

Graham and Rhodes (Graham [1975]) describe a recovery method for precedence parsers that consists of a forward move followed by a correction step. The forward move performs a sequence of reductions on the input list following the error token; they call this the *condensation phase*. Penello [1978] describes a similar condensation step for an LR parser. Both authors give strategies for the correction step.

Feyock and Lazarus [1976] describe a bottom-up system similar to the bounded-range strategy described above. They propose only insertion, deletion, replacement or interchange of terminal symbols as correction strings. Possible strings are filtered in a number of ways, including semantics checks, to yield a set of viable solutions. If the set is nonempty, one is selected through some language-dependent heuristic tests. Their results are impressive; they claim that thrashing is rarely seen.

Mickunas and Modry (Mickunas [1978]) describe an LR(1) recovery system with a forward move and error region patching. Their approach is described in section 12.3.8.

ANNOTATED BIBLIOGRAPHY

Abbreviations:
ACM: Association for Computing Machinery.
CACM: Communications of the Association for Computing Machinery.
IEEE: Institute for Electrical and Electronic Engineers.
IRE: Institute for Radio Engineers.
JACM: Journal of the Association for Computing Machinery.
SIAM: Society of Industrial and Applied Mathematics

Abrahams [1974], P. W., "Some Remarks on Lookup of Structured Variables," *CACM* **17,** (4) 209–210.
Comment on Gates [1973], on lookup of structured Cobol or PL/I variables, pointing out an error in Gates' approach. Alternative methods are also presented that solve the general problem.

Aho [1969a], A. V., J. D. Ullman, "Properties of Syntax Directed Translations," *J. Computer and System Sciences* **3,** (3) 319–334.

Aho [1969b], A. V., J. D. Ullman, "Syntax Directed Translations and the Pushdown Assembler," *J. Computer and System Sciences* **3,** (1) 37–56.
The pushdown assembler (PA) is a multitape stack machine. Given a syntax-directed translator (SDT) with at most k variables on the right part of any one production, then the pushdown assembler requires k + 1 stacks. One stack contains parsing states, and the other k contains strings arising from the syntax-directed transduction. Hierarchy of translators is introduced, and it is shown that the class of PA's is exactly the class of STD's. Rules for constructing the finite control for a given set of productions are given.

Aho [1971], A. V., J. D. Ullman, "Translations on a Context Free Grammar," *Inf. and Control* **19,** 439–475.
Syntax-directed translation, context-free grammars, parse trees, generalized syntax-directed translators, bounds on translation, length relationships, tree automata, conclusions.

Aho [1972a], A. V., J. D. Ullman, *The Theory of Parsing, Translation, and Compiling,* 2 vols., Prentice-Hall, Englewood Cliffs, N.J.
Extensive theoretical framework for languages, grammars, parsers, optimization. (Vol. I) Math preliminaries, introduction to compiling, elements of language theory, theory of translations, general parsing methods, one-pass no backtrack parsing, limited backtrack parsing algorithms. (Vol. II) Techniques for parser optimization, theory of deterministic parsers, translation and code generation, bookkeeping, code optimization. All with formal definitions and many proofs.

Aho [1972b], A. V., J. D. Ullman, "Optimization of Straight Line Programs," *SIAM J. Comput.* **1,** (1) 1–19.
Provides a set of transformations on a sequence of statements into equivalent sequences, then shows that optimization of a straight line sequence under "reasonable" cost criteria can always be accomplished by applying a sequence of these transformations in a prescribed order. Much of this material is also in Aho [1972c].

Aho [1972c], A. V., T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM J. Comput.* **1,** (4) 305–312.
A grammar G for some context-free language is transformed into a grammar G^1 that accepts every sentence in the alphabet of G, by the addition of error productions. A sentence containing one or more errors may be parsed through an Earley parser (see Earley [1970]), and the minimum number of error productions that generates a parse selected. The method operates in $O(n^3)$ steps at worst, where n is the number of tokens in the input sentence.

Aho [1974a], A. V., S. C. Johnson, "LR Parsing," *Computing Surveys* **6,** (2) 99–124.
Grammars, derivation trees, parsers, parser action and go-to tables, parser construction, parsing ambiguous grammars, parser optimization, error recovery. Very readable, with many examples and illustrations.

Aho [1974b], A. V., J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass.

Aho [1975], A. V., S. C. Johnson, J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *CACM* **18,** (8) 441–452
Generating an LR parser for an ambiguous expression grammar, then fixing the table to reflect some desired associativity rules for the operators. Results in a considerably smaller table than is otherwise the case.

Aho [1976], A. V., S. C. Johnson, "Optimal Code Generation for Expression Trees," *JACM* **23,** (3) 488–501.
Algorithms for code generation from expression trees, with theorems.

Aho [1977a], A. V., S. C. Johnson, J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Fourth ACM Symposium on Principles of Programming Languages,* 21–28.
Some conclusions regarding register allocation in single and double length results with even assignment for doubles, the problem of optimal allocation.

Aho [1977b], A. V., S. C. Johnson, J. D. Ullman, "Code Generation for Expressions with Common Subexpressions," *JACM* **24,** (1) 146–160.
Shows that the problem of generating optimal code for expressions containing common subexpressions is computationally difficult, even for simple expressions and simple machines. Some heuristics are given.

Alexander [1975], W. G., "Static and Dynamic Characteristics of XPL Programs," *Computer,* 41–46.
Statistics on 19 XPL programs—distributions of statements by type, parser reductions, operators in expressions, numeric constants, instructional usage (360 target), branch distances.

Allen [1976], F. E., J. Cocke, "A Program Data Flow Analysis," *CACM* **19,** (3) 137–147.
Static analysis methods leading to global data flow analysis. Algorithms given that can determine all the definitions which can reach any node of the control flow graph, and all the live definitions.

Anderson [1973], T., J. Eve, J. J., Horning, "Efficient LR(1) Parsers," *Acta Informatica* **2,** 12–39.

Aufenkamp [1957], D. D., F. E. Hohn, "Analysis of Sequential Machines," *IRE Trans.* EC-6, 276–285.

Backus [1957], J. W., *et al.,* "The FORTRAN Automatic Coding System," *Proc.*

Western Joint Computer Conference **11,** 188–198.
The original definition of the Fortran language and the first automatic translator of Fortran to IBM 704 machine code.

Bauer [1968], H., S. Becker, S. L. Graham, "ALGOL W Implementation," Unpublished paper, Stanford University, Stanford, Calif., Computer Science Dept. CS98.
ALGOL W was designed by Wirth, is a simplified Algol 60 language. This paper describes an IBM 360 implementation, is rich in detail.

Bauer [1974], F. L., "Historical Remarks on Compiler Construction," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 603–621.
Classification of compiler methods and brief historical review of original work and development.

Beatty [1974], J. C., "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM J. Res. Develop.,* 20–39.
An algorithm that permits a high level of optimization at both local and global levels. Finds appreciable improvement over a conventional production compiler. No attempt to assess implementation manpower costs or expected improvements.

Berkeley [1964], E. C., D. G. Bobrow (eds.), *The Programming Language LISP: Its Operation and Applications,* The M.I.T. Press, Cambridge, Mass.
A collection of papers on Lisp: The programming system, styles of programming, applications, implementation, many examples of programs. Among the applications: techniques for automatically discovering interesting relations in data; automation of inductive inference on sequences; machine checking of mathematical proofs; an interpreter for string transformations; a language for an incremental compiler.

Bertsch [1977], E., "The Storage Requirement in Precedence Parsing," *CACM* **20,** (3) 192–194.
A short paper on the compression of precedence tables.

Bochmann [1976], G. V., "Semantic Evaluation from Left to Right," *CACM* **19,** (2) 55–62.
Describes attribute grammars and their use for the definition of programming languages and compilers. Emphasis on attribute conditions that can be fully evaluated in a single pass over the abstract syntax tree.

Booth [1967], T. L., *Sequential Machines and Automata Theory,* Wiley, New York.
A general text on finite-state automata. Complete theoretical development, many examples, applications, practical methods and exercises.

Breuer [1969], M. A., "Generation of Optimal Code for Expressions via Factorization," *CACM* **12,** (6) 333–340.
A complete, but complicated algorithm for finding all factors of a set of expressions to be compiled, then sequencing the operations to minimize the time they need be in memory, then assigning temporary storage cells. Global optimal results are not necessarily obtained.

Brooker [1963], R. A., D. Morris, "The Compiler-compiler," *Ann. Reviews in Auto. Programming,* **3,** Pergamon, Elmsford, N.Y., 229–275.
The first compiler writing system, using a compressed notation, and recursive descent technique.

Bruno [1976], J., R. Sethi, "Code Generation for a One-Register Machine," *JACM* **23,** (3) 502–510.
Shows that generating the optimal code for a one-register machine is hard, i.e., in the same class as the traveling salesman problem.

Brzozowski [1962], J. A., "A Survey of Regular Expressions and Their Applications," *IRE Trans. on Electronic Computers* **11,** (3) 324–335.
One of the early papers on finite-state automata, an exposition of ideas developed by earlier authors. Collects notions of regular expressions, finite-state automata and their transformation.

Campbell [1976], L., *et al.,* "Draft Proposed ANS FORTRAN," *Sigplan Notices,* 11, (3).
Modern draft definition of standard Fortran, with syntax graphs, conventions, semantic specifications.

Chomsky [1956], N., "Three Models for the Description of Language," *IEEE Trans. on Information Theory* **2,** (3) 113–124.
One of the three models in this paper is the phrase-structured grammar. This is the first appearance of this model.

Cocke [1977], J., K. Kennedy, "An Algorithm for Reduction of Operator Strength," *CACM* 20, (11) 850–856.
A simple algorithm that uses an indexed temporary table to perform reduction of operator strength in strongly connected program flow regions. The strength of a multiply in a loop can be reduced to an addition, if one of the factors is constant.

Cohen [1970], D. J., C. C. Gotlieb, "A List Structure Form of Grammars for Syntactic Analysis," *Computing Surveys* **2,** (1) 65–82.
Syntax Graph, graph reductions, compilation from graphs, recursive/nonrecursive compilers, bottom-up parsing with reversed graph.

Conway [1963], M. E., "Design of a Separable Transition-Diagram Compiler," *CACM* **6,** (7) 396–408.
A Cobol compiler design. Accepts a Cobol subset, operates in small memory, requires two working tapes plus a compiler tape. Use of coroutines as modules.

DeMorgan [1977], R. M., I. D. Hill, B. A. Wichman, "A Supplement to the ALGOL 60 Revised Report," *Sigplan Notices* **12,** (1) 52–66.
More clean-up on the Algol report.

DeRemer [1969], F. L., "Practical Translators for LR(k) Languages," Ph.D. thesis, M.I.T., Cambridge, Mass.
Reviews, classifies LR(k) languages and parsers. Introduces SLR grammar (see DeRemer [1971]).

DeRemer [1971], F. L., "Simple LR(k) Grammars," *CACM* **14,** (7) 453–460.
First paper on SLR(k) grammars. These types are generated by an LR(0) construction process, then use FOLLOW(k) sets to resolve inconsistent states.

DeRemer [1974a], F. L., "Transformational Grammars," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 121–145.
A case for language processing as a sequence of tree transformations. Lexical, syntactical processing, standardization, flattening, subtree transformational grammars, extension to regular expressions, example of PL/I declaration defactoring.

DeRemer [1974b], F. L., "Review of Formalisms and Notation," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 37–56.
Concise review of formalisms. Rewriting systems, grammars, Chomsky hierarchy, phrase structures, tree derivation, regular grammars and regular expressions, parsing, parsing strategies, ambiguity, transduction grammars, string-to-tree grammars, self-describing grammars.

DeRemer [1974c], F. L., "Lexical Analysis," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 109–120.
Lexical terms—scanning, screening, characters, tokens, reserved symbols, regularity. Scanner generation via LR construction, handwritten scanner, error recovery, advisability of including conversion routines.

Demers [1975], A. J., "Elimination of Single Productions and Merging Nonterminal Symbols of LR(1) Grammars," *Computer Languages* **1,** 105–119.
Formal treatment of single production removal and merging to optimize LR(1) parser tables, with proofs.

Demers [1977], A., "Generalized Left Corner Parsing," *Fourth ACM Symposium on Principles of Programming Languages,* 170–182.
LC parsing is a combination of LL and LR. Demers develops a parsing machine and demonstrates some minimal properties of the system that facilitate semantic operations through an "anounce point" within each production; the announce point is established by the system as near to the left end of the production formula as possible.

Dijkstra [1976a], E. W. *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N.J.
Explores the thesis that a program should derive from a mathematical statement of a problem in a natural way, thereby automatically resulting in a correct program. Introduces a simple but powerful language suited to the task. Semiformal treatment with many interesting examples.

Dijkstra [1976b], E. W., "On-the-fly garbage collection: an exercise in cooperation," Notes for the 1975 NATO Summer School on Language Hierarchies and Interfaces, in *Lecture Notes in Computer Science,* Springer-Verlag, New York, 46.

Donovan [1972], J. J., *Systems Programming,* McGraw-Hill computer science series, McGraw-Hill, New York.
An introduction to machine structure, machine language, assembly language, assemblers, macros, loaders, programming languages, compilers, and operating systems. IBM System/360 conventions used extensively. The author was associated with project MAC at M.I.T.

Earley [1970], J., "An Efficient Context-free Parsing Algorithm," Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pa. (1968). Also see *CACM* **13,** (2) 94–102.
The general parser—capable of parsing any sentence in any CFG algorithmically. It runs in linear time for a large class of grammars, is bounded by N^2 for unambiguous grammars, and N^3 otherwise. Shows that it is superior to the top-down and bottom-up algorithms of Griffiths and Petrick by empirical studies.

Evans [1964], A., Jr., "An ALGOL 60 Compiler," *Ann. Review in Auto. Programming.* 4, Pergamon, Elmsford, N.Y., 87–124.

Feldman [1966], J. A., "A Formal Semantics for Computer Languages and Application In a Compiler-Compiler," *CACM* **9,** (1) 3–9.
A system that accepts a production language (Floyd) and a special semantics language keyed to the production language, and yields a compiler. Has symbol table operations and a set of mnemonics to identify kinds of expression/statement under consideration.

Feldman [1968], J., D. Gries, "Translator Writing Systems," *CACM* **11,** (2) 77–113.
Review paper on compiler writing systems. Syntax, syntax trees, grammar, operator precedence, precedence, transition matrices, production language, bounded context grammars, DPDA, LR(k), Tixier's recursive functions of regular expressions. Semantics: TMG (top-down 1-pass c writer), Meta, Cogent, etc. Compiler-compilers: FSL, TGS, CC (Brooker-Morris). Meta-assemblers and extendible compilers: Metaplan, Plasma, Xpop, Algol C, large bibliography.

Feyock [1976], S., P. Lazarus, "Syntax-Directed Correction of Syntax" *Software-Practice and Experience,* **6,** 207–219.
An error correction method related to XPL bottom-up system, examples, graph of compiler speed vs. error density.

Fischer [1977], C. N., D. R. Milton, S. B. Quiring, "An Efficient Insertion-Only Error-Corrector for LL(1) Parsers," *Fourth ACM Symposium on Principles of Programming Languages,* 97–103.
Defines a class of insert-correctable LL(1) languages, those for which any error can be corrected by insertion of a suitable terminal string. System will so test a grammar and generate a set of tables that provides optimal string insertion for the correction of syntax errors. Examples.

Floyd [1961], R., "An Algorithm for Coding Efficient Arithmetic Operations," *CACM* **4,** 42–51.
Specialized method for dealing with arithmetic expression groups to reduce code for single-register machine.

Floyd [1963], R. W., "Syntactic Analysis and Operator Precedence," *JACM* **10,** (3) 316–333.
Landmark paper on operator precedence. Has an Algol 60 precedence matrix, theory of operator precedence.

Floyd [1964], R. W., "Bounded Context Syntactic Analysis," *CACM* 7, (2) 62–67.
A bottom-up parsing method based on context analysis of production right parts.

Gates [1973], G. W., David A. Poplawski, "A Simple Technique for Structured Variable Lookup," *CACM* 16, (9) 561–565.
A method for the lookup of structured Cobol or PL/I variables is given. It also checks legality of Cobol identifiers.

Gill [1962], A., *Introduction to the Theory of Finite State Machines,* McGraw-Hill, New York.
An early text on finite-state automata. Discusses classes of automata, equivalence, reduction and reduction algorithms. Semiformal treatment. (But no treatment of regular expressions or regular grammars).

Ginsburg [1962], S., *An Introduction to Mathematical Machine Theory,* Addison-Wesley, Reading, Mass.

Ginsburg [1966a], S., *The Mathematical Theory of Context-Free Languages,* McGraw-Hill, New York.

The first general text on formal context-free language theory. Many general theorems, most original to the author.

Ginsburg [1966b], S., S. Greibach, "Deterministic Context-Free Languages," *Inf. and Control* **9,** (6) 620–648.

Glanville [1978], R. S., S. L. Graham, "A New Method for Compiler Code Generation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* 1978, 231–240.
A construction algorithm for generating machine instructions from a prefix translation of a suitable abstract syntax tree, based on a table representation of the target machine. Conditions for correctness and normal termination of the algorithm are given.

Graham [1975], S. L., S. P. Rhodes, "Practical Syntactic Error Recovery," *CACM* **18,** (11) 639–650.
Error recovery for precedence parsers, experimental results from several different grammars.

Gries [1971], D., *Compiler Construction for Digital Computers,* Wiley, New York.
A compiler construction textbook. Emphasis on practical methods: Algol control structures and expressions. Reviews: grammars and languages, scanner, top-down recognizers, simple precedence, other bottom-up recognizers, production language, run-time storage organization, symbol table organization, internal forms of the source, semantic routines introduction, semantics for Algol constructs, storage allocation, error recovery, interpreters, code generation, code optimization, macros, translator writing systems.

Gries [1977], D., "An exercise in proving parallel programs correct," *CACM* **20,** (12) 921–930.
A parallel program correctness proof method, and its application to an on-the-fly garbage collector (*cf.* Dijkstra [1976], Steele [1975].)

Griffiths [1974a], M., "Run-time Storage Management," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 195–221.
Classical storage allocation and access. Static allocation, dynamic allocation, block linkage, display, stack compaction, parameter linking, labels/go to, aggregates, lists, garbage collection, storage collapse, parallel processes.

Griffiths [1974b], M., "LL(1) Grammars and Analyzers," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 57–84.
Predictive Analysis, LL(1) conditions, decision algorithm, recursive descent construction, grammar transformation, semantic insertion, LL(k) grammars, practical results.

Griffiths [1974c], M., "Introduction to Compiler Compilers," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 356–365.
Brief review of compiler writing systems and current research.

Hall [1974], A. D., private communication, reported in Waite [1974b].

Harrison [1965], M. A. *Introduction to Switching and Automata Theory,* McGraw-Hill, New York.

Harrison [1977], W., "A New Strategy for Code Generation—the General Purpose Optimizing Compiler," *Fourth ACM Symposium on Principles of Programming Languages,* 29–37.

Uses global flow analysis and an intermediate language of simple primitives to achieve general optimization, regardless of source language form.

Hartmanis [1966], J., R. E. Stearns, *Algebraic Structure of Sequential Machines,* Prentice-Hall, Englewood Cliffs, N.J.

Hill [1974], U., "Special Run-time Organization Techniques for Algol 68," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 222–252.
Data storage and management required for Algol 68. Static, dynamic storage, heaps, generative and interpretative handling, special data objects, slicing, rowing, scope checking, scope of procedures, generation of local objects, blocks and procedure calls, actual parameters, garbage collection.

Hoare [1969], C. A. R., An Axiomatic Basis for Computer Programming, *CACM* **12,** 576–581.
Definition and development of the Hoare axiomatic program correctness proof method, e.g., A1 (S) A2; A1 and A2 are assertions regarding the state of a program system, and S is some executable statement. Proof problem is to show that A1 implies A2, given execution of S.

Hoare [1973], C. A. R., N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica* **2,** 335–355.
Brief review of Hoare axiomatic approach, has Pascal statements expressed in axiomatic form, Pascal syntax graphs.

Hopcroft [1969], J. E., J. D. Ullman, "Formal Languages and Their Relation to Automata," Addison-Wesley, Reading, Mass. Text on formal language and automata theory. Largely superseded by Aho [1972].

Horning [1974a], J. J., "What the Compiler Should Tell the User," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 525–548.
Normal output, reaction to errors, syntactic errors, other errors, error diagnosis.

Horning [1974b], J. J., "Structuring Compiler Development," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 498–513.
Goals of compiler development—correctness, availability, generality, adaptability, helpfulness, efficiency. Trade-offs, processes in development: specification, design, implementation, validation, evaluation, maintenance. Management tools: project organization, information distribution and validation, programmer motivation. Technical tools: compiler compilers, standard designs, off-the-shelf components, structured programs, appropriate languages.

Horning [1974c], J. J., "LR Grammars and Analysers," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 85–108.
Intuitive description, definitions of terms, interpreting LR tables, constructing LR tables, representing LR tables, reduction for efficiency, properties of LR grammars and analysers, some grammar modifications to obtain LR, grammar/language inclusions.

Huffman [1954], D. A., "The Synthesis of Sequential Switching Networks," *J. Franklin Inst.* **257,** 161–190, 275–303.

Ingerman [1961], P. Z., "Dynamic Declarations," *CACM* **4,** (42) 59–60.
Algorithm for mapping an OWN array in which dimensions have been

changed—the nontrivial multidimension dynamic array problem is as dealt with here.

Irons [1961], E. T., "A Syntax Directed Compiler for ALGOL 60," *CACM* **4,** (42) 51–55.

Irons [1963], E. T., "An Error-Correcting Parse Algorithm," *CACM* **6,** (11) 669–673. A top-down error-correcting parsing system.

Iverson [1962], K., *A Programming Language,* Wiley, New York.
The original book on APL. Defines the language and gives many examples of its applications to broad areas of computing, e.g., mapping and permutations, ordered trees, graph traversal, microprogramming, matrices, searching, sorting, and the logical calculus.

James [1972], L. R., *A Syntax Directed Error Recovery Method,* Tech. Report CSRG-13, University of Toronto Press, Toronto.
Table-driven error recovery embedded in an LALR(1) parser. This method drops stack states and input symbols, searches for an insertion or compatible state; uses a 2-symbol lookahead limit then 5-symbol lookahead in dire cases, limits stack cutback through a fixed limit.

Johnson [1968], W. L., J. H. Porter, S. I. Ackley, D. T. Ross, "Generation of Efficient Lexical Processors Using Finite State Automatic Techniques," *CACM* **11,** (12) 805–813.
Description of the AED RWORD system that accepts regular expressions and generates a finite-state automaton to recognize the expression's language. Has "escape hatches" to provide for unusual lexical constructs. Used in several different compilers as a lexical analyzer.

Kildall [1973], G. A., "A Unified Approach to Global Program Optimization," *ACM Symposium on Principles of Programming Languages,* 194–206.

Kleene [1952], S. C., *Introduction to Metamathematics,* Van Nostrand Reinhold, New York.

Kleene [1956], S. C., "Representation of Events in Nerve Nets," in C. Shannon and J. McCarthy, *Automaton Studies,* Princeton University Press, Princeton, N.J.

Knuth [1965], D. E., "On the Translation of Languages from Left to Right," *Inf. and Control,* **8,** 607–639.
Original paper on LR(k) languages. Shows that LR(k) languages are equivalent to the deterministic k-symbol lookahead languages, gives two parser construction methods, and proves that the viable prefix set is a regular language.

Knuth [1968], D. E., *Fundamental Algorithms,* Vol. 1 of *The Art of Computer Programming,* Addison-Wesley, Reading, Mass.

Knuth [1971], D. E., "An Empirical Study of FORTRAN Programs," *Software—Practice and Experience* **1,** 105–134.

Kohavi [1971], Z., "Switching and Finite Automata Theory," McGraw-Hill, New York.

Korenjak [1969], A. J., "A Practical Method for Constructing LR(k) Processors," *CACM* **12,** (11), 613–623.
Large grammar is partitioned into several smaller parts, each of which is parsed independently; the mechanism of parser intercommunication and reporting is developed.

Koster [1974], C. H. A., "Using the CDL Compiler-Compiler," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 366–426.
Detailed review of the CDL compiler writer, a top-down system with symbol table manager system, macros, etc.

Kurki-Suonio [1969], "Notes on Top-Down Languages," *BIT* **9,** 225–238.
A short paper on the properties of LL(k) languages and grammars.

LaFrance [1971], J. E., "Syntax Directed Error Recovery for Compilers," Ph.D. thesis, University of Illinois, Urbana, Computer Science Dept. ILLIAC IV Doc. 249.

LaLonde [1971a], W. R., E. S. Lee, J. J. Horning, "An LALR(k) Parser Generator," *Proc. IFIP Congress* **71,** TA-3, North-Holland Pub. Co., Netherlands, 153–157. See LaLonde [1971b]. Amsterdam.

LaLonde [1971b], W. R., "An Efficient LALR Parser Generator," Tech. Report CSRG-2, University of Toronto Press, Toronto.
Review of LR machines, introduction of LALR(1) parsing algorithm—this finds lookahead sets for inconsistent states based on LR(0) machine and state tracing. However, DeRemer has shown that LaLonde's system actually accepts a subset of the LALR(1) languages.

Lampson [1977], B. W., *et al.,* "Report on the Programming Language Euclid," *Sigplan Notices* **12,** (2).
Euclid draws heavily upon Pascal for its structure and many of its features. The intention is to express programs that are to be verified by formal methods.

Ledgard [1975], H. F., M. Marcotty, "A Genealogy of Control Structures," *CACM* **18,** (11) 629–639.
Review and classification of known control structures, discussion of equivalence and reducibility, hierarchy, examples, four general conclusions.

Lee [1967], J. A. N., *Anatomy of a Compiler,* Reinhold, New York.

Leinius [1970], R. "Error Detection and Recovery for Syntax-Directed Compiler Systems," Ph.D. thesis, University of Wisconsin, Madison.
Mostly simple precedence error recovery treatment, one chapter on LR(k) systems. Uses EULER as model language. Method requires additional states in the LR(k) parser, created heuristically.

Levy [1975], J. P., "Automatic Correction of Syntax-Errors and Programming Languages," *Acta Informatica* **4,** 271–292.
Formal models of error correction. Notign of error, global error correction, local error correction, detailed error correction method for one language construct, problems, practical error correction.

Lewis [1968], P. M., R. E. Stearns, "Syntax-Directed Transduction," *JACM* **15,** (3) 465–488.
Transduction grammars, relation to LR and LL grammars, machines.

Lewis [1971], P. M., D. J. Rosenkrantz, "An ALGOL Compiler Designed Using Automata Theory," *Proc. of the Polytechnic Inst. of Brooklyn, New York, Symposium on Computers and Automata,* New York, 1971. Polytechnic Press of the Polytechnic Institute of Brooklyn; Distributors: Wiley-Interscience, New York. 75–87.

Loveman [1977], D. B., "Program Improvement by Source-to-Source Transformation," *JACM* **24,** (1) 121–145.
User-provided assertions in an Algol program are shown to be valuable in optimization.

Maley [1963], G. A., J. Earle, *The Logic Design of Transistor Digital Computers,* Prentice-Hall, Englewood Cliffs, N.J.

Marcotty [1976], M., H. F. Ledgard, G. V. Bochman., "A Sampler of Formal Definitions," *Computing Surveys* **8,** (2) 191–276.
Presentation of four well-known formal definition techniques: W-grammars, production systems with an axiomatic approach to semantics, the Vienna definition language, and attribute grammars. Each technique is described tutorially and examples are given; then each is applied to define the same small programming language.

Maurer [1975], W. D., T. G. Lewis, "Hash Table Methods," *Computing Surveys* **7,** (1) 5–19.
Hashing functions, collision, bucket overflow, alternatives to hashing. Limited analysis of efficiency.

McCarthy [1960], J., "Recursive Functions of Symbolic Expressions and their Computation by Machine," *CACM* **4,** (4) 184–195.
The original paper on the mathematical basis of LISP. Five primitive recursive functions are shown to form the basis of a complete programming language. Interpretation of structures as directed graphs.

McCarthy [1962], J., *et al., LISP 1.5 Programmer's Manual,* The M.I.T. Press, Cambridge, Mass.
A programmer's manual for LISP. Some examples, but best used as a reference document.

McCluskey [1965a], E. J., *Introduction to the Theory of Switching Circuits,* McGraw-Hill, New York.

McCluskey [1965b], E. J., T. C. Bartee, *A Survey of Switching Circuit Theory,* McGraw-Hill, New York.

McCullough [1943], W. S., E. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bull. of Math. Biophysics* **5,** 115–133.
An original paper on finite-state automata and their relation to regular expressions.

McKeeman [1970], W. M., J. J. Horning, D. B. Wortman, *A Compiler Generator,* Prentice-Hall, Englewood Cliffs, N.J.
A text on compiler construction. Introduction to formalism, LR(k) and precedence parsing. Emphasis on MSP (mixed strategy precedence) method. Contains complete compiler generator programs in XPL and a definition of XPL as a language.

McKeeman [1974a], W. M., "Compiler Construction," in *Lecture Notes in Computer Science,"* G. Goos and J. Hartmanis, Springer-Verlag, New York, 1–36.
Broad informal review of compiler components. Definitions, source/target language, implementation language, recursive descent compilers, modularization, specification, feedback-free, vertical/horizontal fragmentation, transformations.

McKeeman [1974b], W. M., "Symbol Table Access," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 514–524.
Review of linear, sorted, tree, and hash symbol table access methods. Block-structured symbol tables. Contains complete XPL programs and sample traces. Hash functions, secondary stores, evaluation of access methods.

McKeeman [1974c], W. M., "Programming Language Design," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 514–524.
Who should or should not design languages? Design principles, models for languages: street language, the Algol family.

McNaughton [1960], R., H. Yamada, "Regular expressions and State Graphs for Automata," *IRE Trans on Elect. Computers* **9,** (1) 39–47. Reprinted in R. Moore, *Sequential Machines: Selected Papers,* Addison-Wesley, Reading, Mass., (1964).
First paper giving algorithms for interconverting state graphs, regular expressions. Deals with all possible regular expressions—union, intersection, complement, closure, concatenation. Theorems and proofs given.

Mealy [1955], G. H., "Method for Synthesizing Sequential Circuits," *Bell System Tech. J.* **34,** 1054–1079.

Metcalfe [1964], H. H., "A Parameterized Compiler Based on Mechanical Linguistics," *Ann. Reviews in Auto. Programming* **4,** Pergamon Press, 125–165.
A descriptive paper on some recursive descent programming techniques, with special mechanisms for semantic operations. Well suited to a top-down string translator. Informal. Brief discussion of the validation problem (verifying LL(1)), but no solution.

Meyers [1965], W. J., "Optimization of Computer Code", unpublished memorandum, G. E. Research Center, Schenectady, N.Y., 12 pages.

Mickunas [1976], M. D., R. L. Lancaster, V. B. Schneider, "Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars," *JACM* **23,** (3) 511–533.
Grammar transformation results and algorithms.

Minsky [1967], M., *Computation: Finite and Infinite Machines,* Prentice-Hall, Englewood Cliffs, N.J.
Textbook on finite state automata. Many unusual and interesting examples and side issues.

Moore [1956], E. F., "Gedanken-Experiments on Sequential Machines," in *Automata Studies,* Princeton University Press, Princeton, N.J., 129–153.

Morgan [1970], H. L., "Spelling Correction in System Programs," *CACM* **13,** 90–94. A spelling correction algorithm and its applications.

Morris [1968], R. "Scatter Storage Techniques," CACM VII, (1), 38–43.

Nakata [1967], Ikuo, "On Compiling Algorithms for Arithmetic Expressions," *CACM* **10,** (8) 492–494.

Naur [1963], P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *CACM* **6,** (1) 1–17.
The original Algol 60 definition.

Pager [1979], D. "A Practical General Method for Constructing LR(k) Parsers," Acta Informatica **7,** 249–68.

Paul [1962], M., "Zur Struktur formaler Sprachen," Universitat Mainz, Dissertation D77.

Paul [1962a], M., "A General Processor for Certain Formal Languages," *Proc. of the Symposium on Symbolic Languages in Data Processing,* Rome, 1962. Also: Gordon and Breach, New York, 1962, 65–74.

Pennello [1978], T. J., F. DeRemer, "A Forward Move Algorithm for LR Error Recovery," *Fifth Annual ACM Symposium on Principles of Programming Languages,* Jan. 23, 241–254.
A "forward move" is useful in syntax error recovery. In a forward move, the text past an error token is partially reduced in an attempt to develop information useful in patching over the error. Pennello and DeRemer develop this for an LR parser.

Perlis [1956], A. J., *et al.,* "Internal translator (IT), a Compiler for the 650," Carnegie Institute of Technology, Computation Center, Pittsburgh. Also: Lincol Lab. Div. 6, Doc. 6D-327.

Poole [1974], P.C., "Portable and Adaptable Compilers," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 427–497.
Survey of issues of transporting languages and compilers. Problems with current compilers, standards. Survey of techniques: high-level language coding, bootstrapping, language-machine interface, abstract machine modeling, Janus. Case studies: AED, LSD, BCPL, Pascal, IBM Fortran G.

Purdom [1974], P., "The Size of LALR(1) Parsers," *BIT* **14,** 326–337.
Some statistical results enabling prediction of the size of LALR parser tables from some simple grammar measures, based on a large set of grammars.

Randell [1964], B., L. J. Russell, *ALGOL 60 Implementation,* Academic Press, New York.
Contains a detailed machine model for ALGOL 60 implementation. Review of nested block structure, procedure calls, parameter passing, branching, control structures, etc.

Rosen [1967], S. (ed.), *Programming Systems and Languages,* McGraw-Hill, New York.
An early collection of papers on languages, compilers, macros.

Rosen [1973], B. K., "Tree-Manipulating Systems and the Church-Rosser Theorems," *JACM* **20,** (1) 160–187.
Considers subtree replacement systems (RPS) and develops sufficient conditions for the Church-Rosser property; indicates that the result of a set of subtree replacements under some replacement system RPS is independent of the order of the replacement. This property does not hold for all tree RPS. Applications to recursive definitions, the lambda calculus and McCarthy's recursive calculus are discussed.

Rosenkrantz [1970a], D. J., P. M. Lewis, "Deterministic Left-Corner Parsing," *IEEE Conf. Record 11th Annual Symposium on Switching and Automata Theory,* 139–152.
The original left-corner paper. See Demers [1977].

Rosenkrantz [1970b], D. J., R. E. Stearns, "Properties of Deterministic Top-Down

Grammars," *Inf. and Control* **17,** 226–256.
Definitions, Test for LL(k), strong LL(k), e-rules, canonical push-down automata, LL(k) hierarchy, equivalence decidability, properties.

Sammet [1969], J. E., *Programming Languages: History and Fundamentals,* Prentice-Hall, Englewood Cliffs, N.J.
A catalog and brief introduction to the reported programming languages known at that time.

Sammet [1976], J. E., "Roster of Programming Languages for 1974–5," *CACM* **19,** (12) 655–669.
A sorted list of 167 languages, each described by a dozen lines or so related to availability, implementation, etc. Also see Sammet [1969].

Sattley [1961], K., "Allocation of Storage for Arrays in ALGOL 60," *CACM* **4,** (42) 60–65.
General problem of dynamically allocating array space for ALGOL 60.

Sethi [1970], R., J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *JACM* **17,** (4) 715–728.
Given: a machine with $N \geq 1$ general purpose registers, arithmetic instructions that may operate between register/register or register/memory (e.g., IBM 360), and arithmetic expressions. No optimization for common subexpressions. Then two optimal register assignment algorithms are given: one in which no algebraic properties are assumed and another in which certain operators are commutative or both commutative and associative. These algorithms are shown to require a minimal number of storage references in the evaluation and a minimal number of instructions.

Shamir [1971], E., "Some Inherently Ambiguous Context-Free Languages," *Inf. and Control* **18,** 355–363.
Introduces a class of inherently ambiguous CFGs.

Sigplan [1973], "Proceedings of a Symposium on High-Level-Language Computer Architecture," *Sigplan Notices* **8,** (11).
Nineteen papers on this subject.

Sigplan [1974], "Proceedings of a Symposium on Very High Level Languages," *Sigplan Notices* **9,** (4).
Fifteen papers on high-level languages.

Sigplan [1975a], "Programming Language Design," *Sigplan Notices* **10,** (7).
Nine short papers on language design. Structured control, data types and program correctness, extensibility, structured languages, abstract data types, exception handling issues, cognitive psychology, and programming language design.

Sigplan [1975b], "1975 International Conference on Reliable Software," *Sigplan Notices* **10,** (6).
Sixty-three papers in this area; three are related to languages and their influence on reliable software.

Sigplan [1976], "Interface Meeting on Programming Systems in the Small Processor Environment," *Sigplan Notices* **11,** (4).
Twenty-four papers on this subject; four are related to programming languages for small processors.

Soisalon-Soininen [1977], E., "Elimination of Single Productions from LR Parsers

in Conjunction with the Use of Default Reductions," *Fourth ACM Symposium on Principles of Programming Languages,* 183–193.
Review of LR construction, development of method for elimination of single production transitions, with concomitant reduction of table size.

Stearns [1967], R. E., "A Regularity Test for Pushdown Machines," *Inf. and Control* **11,** (3) 323–340.
An algorithm for determining whether a given context-free grammar is regular.

Stearns [1969], R. E., P. M. Lewis, "Property Grammars and Table Machines," *Inf. and Control* 14, 524–549.
Formal definition of an attribute system for context free grammars; attributes are effectively attached to derivation tree nodes and used to control bindings and legal derivations.

Steele [1975], G. L., Jr., "Multiprocessing compactifying garbage collection," *ACM* **18,** (9) 495–508. On-the-fly garbage collector. Also see Dijkstra [1976] and Gries [1977].

Tanenbaum [1976], A. S., "A Tutorial on ALGOL 68," *Computing Surveys* **8,** (2) 155–190.
Very readable introduction to ALGOL 68, with many examples and good discussion.

Van Wijngaarden [1969], A. (ed.), "Report on the Algorithmic Language ALGOL 68," *Numerische Mathematik* **14,** 79–218.
Definition of ALGOL 68.

Waite [1974a], W. M., "Assembly and Linkage," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 333–355.
Model for assembly. Object and statement procedures, cross-referencing, back chaining, storage constraints, two-pass assembly, the RESERVE expression problem, partial assembly and linkage.

Waite [1974b], W. M., "Optimization," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 549–602.
Classification of techniques: transformations, regions, efficacy. Local optimization: expression rearrangement, redundant code elimination, basic blocks. Global optimization: redundancy and rearrangement, frequency and strength reduction, global analysis.

Waite [1974c], W. M., "Relationship of Languages to Machines," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 170–194.
Considerations in selecting a suitable interface between a language and a target machine. Data objects: encodings, interpretations, primitive/derived modes, mode conversion, formation rules. Register structure, data access, aggregates, procedures, procedure calls.

Waite [1974d], W. M., "Semantic Analysis," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 157–169.
Discussion of what to do after abstract syntax tree is formed. Optimizing transformations, attribute propagation, flattening through traversals. Postfix vs. prefix. Operator identification and coercion, semantic ambiguity.

Waite [1974e], W. M., "Code Generation," in *Lecture Notes in Computer Science,* G. Goos and J. Hartmanis, Springer-Verlag, New York, 302–332.

A model of code generation, based on Wilcox [1971]. Contains a transducer, simulator. Handles common subexpression optimization.

Warshall [1962], S., "A Theorem on Boolean Matrices," *JACM* **9,** (1) 11–12. A fast algorithm for computing the transitive closure of a relation, along with a proof. (Also proven in a different way in Aho [1972]).
Prefix- and postfix-translatable objects. Generator data structures, value descriptors, register descriptors, instruction generation, primitive operations, interpretative coding languages. Target of code generation is some assembly language.

Whitney [1969a], G., "An Extended BNF for Specifying the Syntax of Declarations," *AFIPS* Spring Joint Computer Conference, 801–812.
Formal symbol table functions and grammar extensions, suitable for top-down compiler system for block-structured grammar. Example language (MAL).

Whitney [1969b], G., "The Generation and Recognition Properties of Table Languages," *Information Processing* **68,** North-Holland, 388–394.
A formal paper on his table language generators, a table automaton, and five closure properties.

Wirth [1966], N., H. Weber, "EULER: A Generalization of ALGOL, and its Formal Definition," Part 1, *CACM* **9,** (1) 13–25; Part 2, *CACM* **9,** (2) 1966, 89–99.
(Part 1) Elementary notation for algorithms, phrase structure grammar, simple precedence, precedence matrix, higher-order precedence. (Part 2) Euler language: precedence matrix and functions, language definition, productions, interpretation of operators, examples.

Wirth [1971a], N., "The Programming Language PASCAL," *Acta Informatica* **1,** 35–63.
First published definition of Pascal, semiformal.

Wirth [1971b], N. "The design of a PASCAL compiler," *Software—Practice and Experience* **1,** 309–333.
A description of the design of a PASCAL compiler for the CDC 6000 series, including symbol table structure details, organization of the project, various statistical results on instruction usage, nesting levels, etc. Most of the article is applicable to a Pascal implementation on any machine.

Wirth [1976a], N., "Programming Languages: What to Demand and How to Assess Them," *Symposium on Software Engineering,* Belfast. What language should do for the user; a strong case for Pascal.

Wirth [1976b], N., "Professor Cleverbyte's Visit to Heaven," private communication.
A tongue-in-cheek tale of a heaven in which every possible feature of every possible language is implemented in a colossal computer, with such a large operating system that it breaks down 50 times per second (but recovers through elaborate mechanisms).

Wirth [1976c], N., *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, N.J.
PASCAL textbook. Applications to data structures, files, sorting, recursive algorithms, dynamic information structures, language structures and compilers. Latter chapter contains a complete top-down definition of grammar and construction of a compiler for a small language (PL/O), including transformation of syntax graphs into program structures, checking for validity, error recovery, scanning, code generation for an Algol-class stack machine.

INDEX

REORDER NUMBER 13-4160