

# MEMO

CONTROL DATA  
CANADA LTD.

DATE: 12 May 1975

TO: DISTRIBUTION

LOCATION:

EXT:

FROM: R. Rothstein

LOCATION: TTOFAC

EXT:

SUBJECT: ISWL TRAINING GUIDE

Attached you will find the ISWL TRAINING GUIDE. This is the first release of this document. The material reflects the current status of the ISWL compiler.

When SWL becomes available this document will be upgraded to reflect the features of SWL. Your comments and criticism will be helpful in improving the quality of this document.

Please do NOT reproduce this document. Additional copies may be obtained from Dave Dudley, PGAASL, {612-830-6419}.

Thank you for your co-operation.



Ron Rothstein,  
IPL Training,  
Canadian Development Division,  
Control Data Canada Limited,  
1855 Minnesota Court,  
Mississauga, Ontario

{416 - 826-8640 X241 or X457}

NOTE: This copy contains Revision B,  
dated 6/30/75.

**INTERIM**

**SOFTWARE WRITERS LANGUAGE**

**TRAINING GUIDE**

**NCR\CDC PRIVATE**

THIS DOCUMENT CONTAINS INFORMATION PROPRIETARY  
TO THE NATIONAL CASH REGISTER COMPANY AND CONTROL  
DATA CORPORATION. ITS CONTENTS SHALL NOT BE  
DIVULGED OUTSIDE OF EITHER COMPANY NOR REPRODUCED  
WITHOUT EXPLICIT PERMISSION OF THE DIRECTOR AND  
GENERAL MANAGER, NCR/CDC ADVANCED SYSTEMS LABORATORY.

**RON ROTHSTEIN**

**JUNE, 1975**





## PREFACE

The growing interest in producing quality software has fostered many schools of thought and many alternative methods of accomplishing this goal. Control Data Corporation, National Cash Register Company, and the Advanced Systems Laboratory have chosen the programming language SWL (Software Writer's Language) as a vehicle for producing quality software.

This text explains ISWL (Interim Software Writers Language) in a tutorial manner. It is not a reference document. Where possible, the reasoning behind certain language features is examined and explained. SWL will be available in the near future and this document will be upgraded at that time to reflect the added features available to the software writer.

Much of the material in this text comes from the questions posed in the many SWL classes which have been taught to-date. Certain portions of this material have been influenced greatly by a few individuals. John Sutherland provided much of the coding and ideas behind "Structured Programming and SWL" as contained in Chapter 5. John Dirnberger put together the extensive "Bibliography" which appears in Appendix D. Alex Seggle provided the initial impetus for producing the "CHARACTER CODES" in Appendix G. In addition I would like to thank the many colleagues who advised, discussed, suggested, and influenced the writing of this material.

Ron Rothstein  
TORONTO, CANADA  
APRIL, 1975

Your comments on this document will be appreciated. Please address all correspondence to:

Purveyor of the ISWL  
Training Guide  
Canadian Development Division  
Control Data Canada Limited  
1855 Minnesota Court  
Streetsville, Mississauga  
Ontario, Canada L5N 1K7



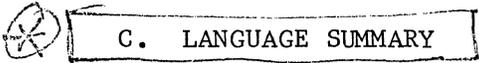
## CONTENTS

### CHAPTER:

INTRODUCTION

1. CONCEPTS OF SWL
2. ELEMENTARY SWL
3. SWL DATA STRUCTURES
4. ADVANCED SWL
5. STRUCTURED PROGRAMMING AND SWL
6. SWL PROGRAMMING TECHNIQUES AND CONVENTIONS
7. PERFORMANCE MEASUREMENT & PREDICTION

### APPENDIX:

- A. RESERVED WORD LIST
- B. ERROR LIST
- C. LANGUAGE SUMMARY
- D. BIBLIOGRAPHY
- E. ANNOTATED TERMINAL SESSION
- F. COMPILATION LISTING
- G. CHARACTER CODES
- H. SWL BNF (ALPHABETICAL)

### LIST OF FIGURES:

### INDEX:



## INTRODUCTION

The Advanced Systems Laboratory (ASL) was founded in September, 1973 as a joint venture of Control Data Corporation (CDC) and the National Cash Register Company (NCR). The goal of ASL is "... to design a line of architecturally compatible computer systems with a broad range of processing power, enabling both companies to meet their respective customers' needs". This new computer line has been christened IPL (Integrated Product Line)<sup>1</sup>.

The method chosen to insure compatibility consists of a high-level implementation language common to all processors in the product line. This language is known as The Software Writer's Language (SWL). Product set members (i.e. FORTRAN, COBOL, Operating System, etc.) for the IPL machines will be written in SWL. In this way, a software product (such as FORTRAN) will be written once in SWL for all IPL machines.

In addition, concern has been expressed regarding the quality, reliability, maintainability, etcetera of software. SWL will provide an effective vehicle for writing software in a uniform, structured manner. This will certainly enhance the quality of IPL software and the ability to maintain and modify that software.

This text is intended to be tutorial. It is expected that the reader have some basic understanding of computer programming. From this base the text provides the building blocks for helping the reader become a competent SWL programmer. You will be exposed first to the concepts underlying this language (i.e. What is?... block structure, pointers, procedures, recursion, stacks, etc.). Next, we concentrate on writing simple SWL programs. Then, a complete discussion of SWL data structures and the more advanced language constructs are presented. Finally, to improve your ability to program the text includes chapters on writing structured programs, applying IPL conventions and programming techniques, and measuring and predicting SWL program performance.

This text may be read from "cover to cover". It is not a reference manual and care has been taken to provide continuity between chapters. Some of the chapters are used in conjunction with SWL courses and may be read separately.

<sup>1</sup> CDC MEMO: "FORMATION OF ADVANCED SYSTEMS LABORATORY"  
R. M. PRICE, PRESIDENT  
CONTROL DATA SYSTEMS & SERVICES COMPANY  
23 AUGUST 1973.



## CHAPTER 1

### CONCEPTS OF THE SOFTWARE WRITERS LANGUAGE

This chapter will explain the most important concepts of SWL. SWL syntax is not included. Check the list of topics below. If you are familiar with all the concepts listed - skip this chapter.

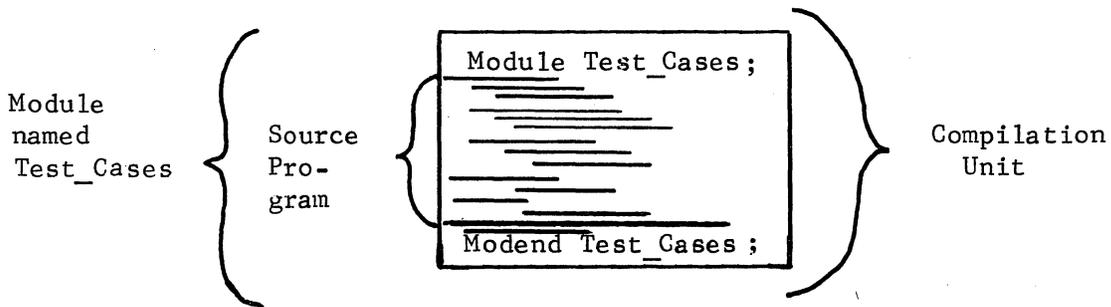
#### CONCEPTS DISCUSSED IN THIS CHAPTER

- Compilation Unit
- Block Structure
  - Scope of Identifiers
  - Procedure
  - Block
- Procedures and Functions
  - Calling Mechanisms
  - Shielding and Sharing Variables
  - Parameter Passing
  - Recursion
- Pointers
- Storage Management
  - Stack
  - Sequence
  - Heap
- Type
- Variables
  - Scope
  - Automatic/Static
- Synchronous/Asynchronous Processing
- Understanding Backus - Naur Form (BNF)

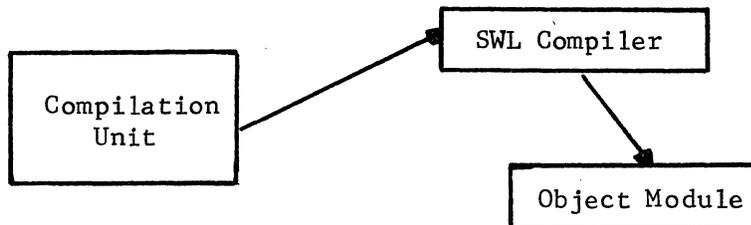
## THE COMPILATION UNIT

To compile any SWL program(s) the compilation unit must be defined. This unit (of compilation) is called a MODULE. The module declaration may (optionally) contain the name or identifier of the module. This module name becomes a most convenient way of referring to the program. The end of the compilation unit must also be defined. A pictorial representation of a compilation unit is given in figure 1.1. Remember, one and only one compilation unit can be compiled at one time. The compilation process transforms a single compilation unit (of source statements) into a single OBJECT MODULE. The compilation process is depicted in Figure 1.2. If five compilation units are to be transformed into five object modules, then five compilation processes must be used.

Note that the compilation unit and module are not synonymous. A module may be used in many ways which will be covered later. One use of the module is to define a compilation unit.



THE COMPILATION UNIT - FIGURE 1.1

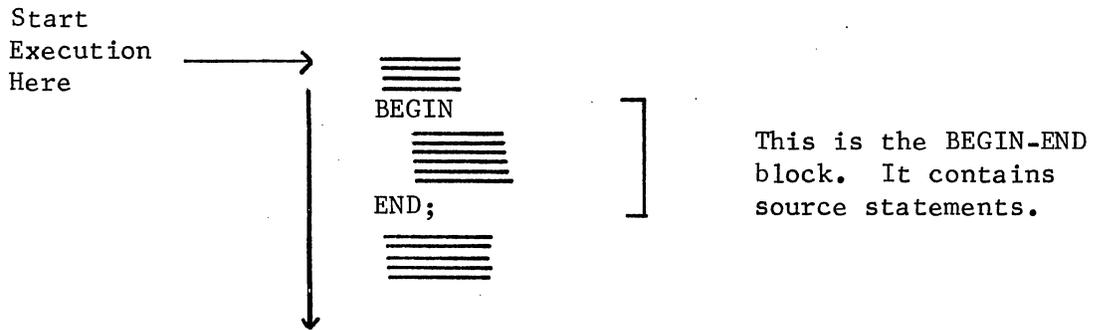


THE COMPILATION PROCESS - FIGURE 1.2

BLOCK STRUCTURE

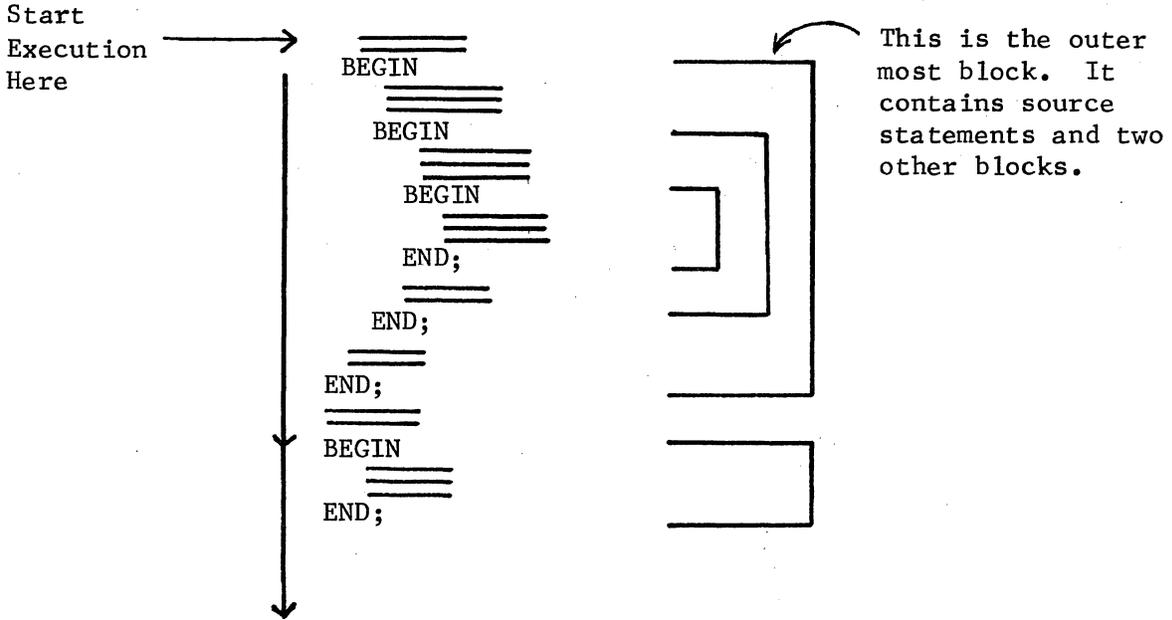
A programming Language is said to be "block structured" when we can identify and create unique blocks (or groups) of source statements and when we can associate a lifetime with the declarations in a block. A block (and hence block structure) in SWL can be identified in one of two ways: 1) BEGIN-END BLOCKS, and 2) PROC-PROCEND BLOCKS.

The BEGIN-END Block is simply a list of source statements surrounded by BEGIN and END. See figure 1.3 below.



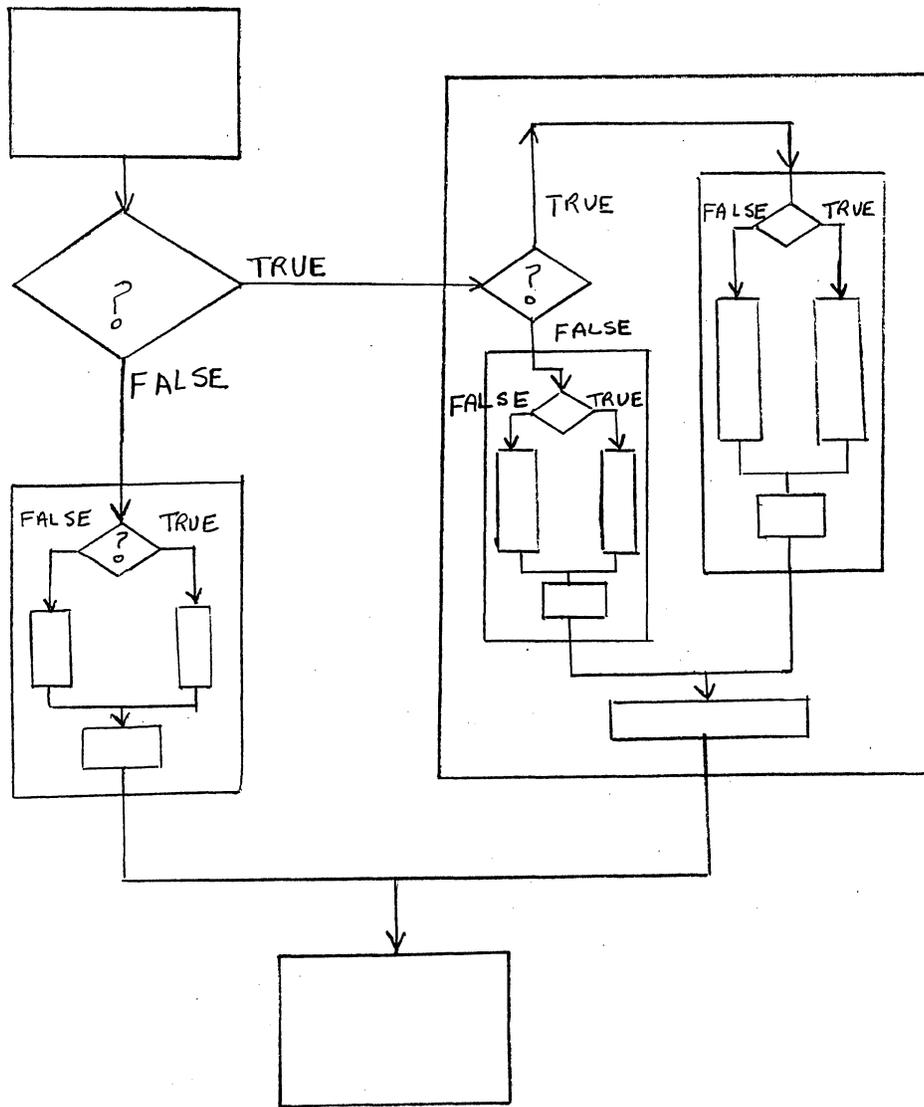
BEGIN-END BLOCK - FIGURE 1.3

Of course, a given block may contain other blocks as shown in figure 1.4.



NESTED BEGIN-END BLOCKS - FIGURE 1.4

You will notice that BEGIN-END Blocks are executed as they are encountered. In the absence of control statements the program will execute from top to bottom. Of what significance is all this structure? The Block structure enables us to easily identify the structure of the program. For example, a block may be written that reads input data, or defines what execution takes place when variables A and B are equal. Implicit here is the notion that a block may be easily replaced by another block whose function is similar but is improved in some way. This concept of "Replaceable blocks" is illustrated in figure 1.5.

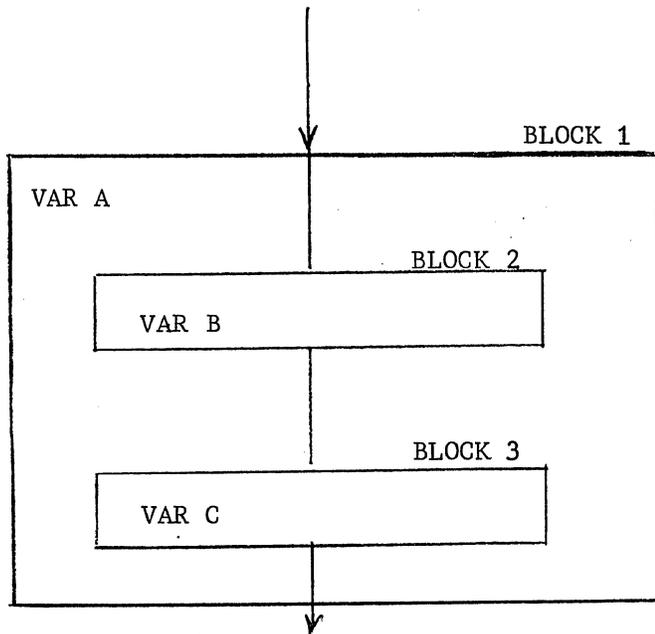


BLOCK STRUCTURE - FIGURE 1.5

Here we see (in figure 1.5) that we can have blocks within blocks. The BEGIN-END statements are the SWL way for implementing this block structure. Also note the help that this block structure provides in implementing software. We can write our software at an overview level first and then go back and amplify the details of each "block".

Also note in Figure 1.5 that during execution only certain blocks will be used. For example: if the boolean test is False only one relatively small block is used. It would be nice if our programming language could utilize this knowledge of BLOCKS to reduce the amount of storage required at execution time, for example. In SWL, variables are declared with the VAR statement. Our BEGIN-END block then can take the form shown in Figure 1.6.





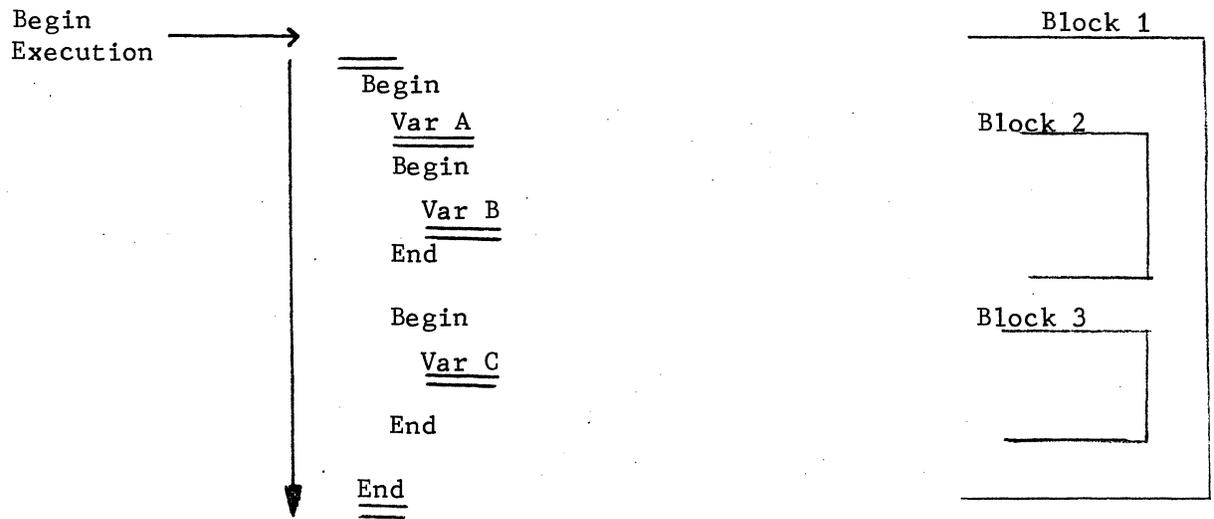
SCOPE OF IDENTIFIERS - FIGURE 1.7

In figure 1.7 we see three blocks. Notice that Block 1 contains Block 2 and Block 3. Each block declares one variable (it may be an array of 1000 integers). The declaration of variable A is local to block 1 but global to blocks 2 and 3. Statements within a block may make valid references to all variables local to their block and also to all variables global to their block. So, for example, statements in Block 2 can reference the variable A and so can statements in Block 3.

Block 3 declares variable C to be local. Hence statements in Block 3 can reference variables C and A. (because A is global to block 3). However, there can be no reference to variable B from statements within Block 3. This is because the variable B is neither Local or Global to Block 3.

In general, then, we try to declare variables in such a way as to make them available when needed.

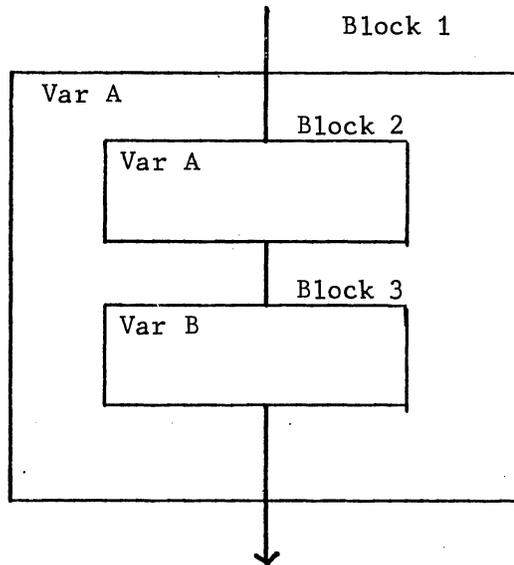
Figure 1.7, when written in SWL would appear as shown in figure 1.8.



SCOPE OF IDENTIFIERS (SWL) - FIGURE 1.8

Of course, we could place all the variable declarations in Block 1 and our program would run just fine, but we might be wasting storage space during the program execution.

The final question to be asked is: what happens when variable names conflict as in figure 1.9?



IDENTIFIER CONFLICTS - FIGURE 1.9

If figure 1.9 the variable A is declared Local in blocks 1 and 2. The result is that any reference to variable A from within blocks 1 or 3 will actually refer to the A declared in Block 1. However, when Block 2 is entered at execution time a new (local) variable named A will be created. This new variable A will not destroy the old (Block 1) variable A. Of course, when we exit Block 2 its local variable A is removed and then only the A declared in Block 1 exists.

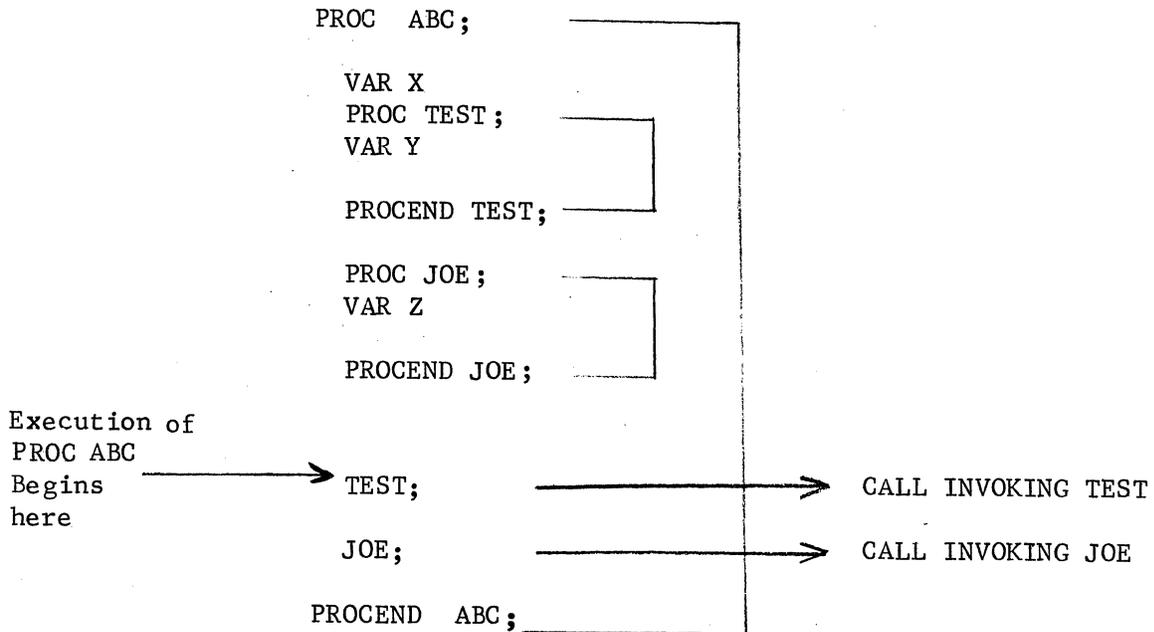
The second method for declaring block structure is the procedure (PROC). Procedures are declared with the PROC-PROCEND statements as bounds on the procedure. Like BEGIN-END blocks, Procedure blocks may have global and local variables. Procedures may contain Procedure blocks and Procedure identifiers are shared and shielded exactly as variable identifiers are. Unlike BEGIN-END blocks, the Procedure block is called by referring to the name of the procedure.

Figure 1.10 shows a simple Procedure block.



A SIMPLE PROCEDURE BLOCK - FIGURE 1.10

Of course, procedure blocks may contain other procedure blocks as shown in Figure 1.11.



NESTED PROCEDURE BLOCKS - FIGURE 1.11

In figure 1.11, Procedure ABC contains procedures TEST and JOE. With this organization TEST and JOE are SHIELDED by ABC. That is, procedures outside ABC may not call TEST or JOE directly. Only statements within ABC may call TEST or JOE.

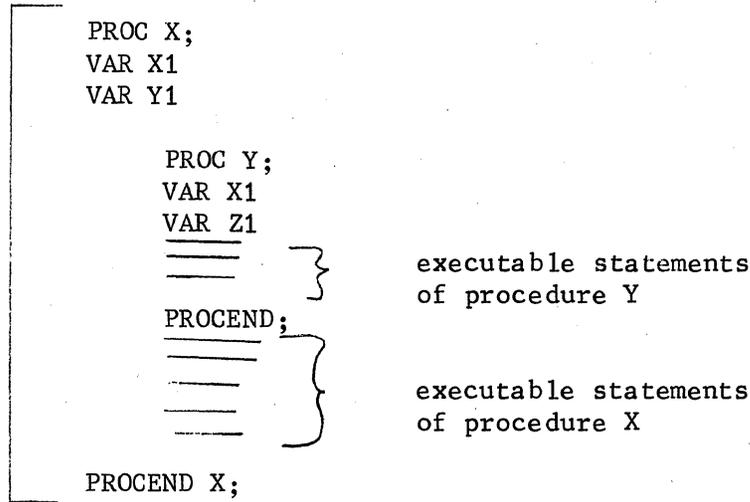
Any variables declared in ABC (such as X above) are global to TEST and JOE. Variables declared in either TEST or JOE (such as Y & Z) are local to their respective procedures.

Finally, BEGIN-END Blocks may be declared within any of these procedures.

In summary, we have examined the concept of block structure. We have seen how blocks may be created either through BEGIN-END or PROC-PROCEND statements. The sharing and shielding of variables was discussed in relation to both types of blocks.



Second, procedures provide the ability of shielding and sharing variables. Variables which are declared inside a procedure are called local variables. Their scope is the procedure (block) in which they are defined. See figure 1.15.



SHIELDING & SHARING VARIABLES - FIGURE 1.15

The shielding and sharing of variables declared in procedures is very similar to the BLOCK method of shielding and sharing variables. In figure 1.15 variable Y1 is local to Procedure X. Since Procedure Y is contained in Procedure X variable Y1 may also be referenced in Procedure Y. In this case (from the point of view of Procedure Y), we say that variable Y1 is global to Procedure Y. Similarly, variable Z1 declared in Procedure Y is local to Procedure Y and cannot be referenced in Procedure X. In the case of variable X1, we really have two local variables. One variable, X1 declared in Procedure X, is local to Procedure X. The variable X1 declared in Procedure Y, however, denotes a local variable which can be referenced only in Procedure Y. This is because the identifier X1 appears in both procedures. Figure 1.16 clarifies some of these points.

VARIABLE IDENTIFIER	Declared in Procedure	Can Variable be referenced in Procedure X?	Can Variable be referenced in Procedure Y?	Scope of the Variable Identifier
X1	X	YES	NO	Procedure X
Y1	X	YES	YES	Procedures X & Y
X1	Y	NO	YES	Procedure Y
Z1	Y	NO	YES	Procedure Y

SCOPE OF VARIABLE IDENTIFIERS - FIGURE 1.16

When a procedure is called we often "pass parameters" to the procedure. These parameters consist of data that the called procedure needs to operate correctly. For example, suppose we had a procedure that would sort arrays of integers. Typically, we would call the sort procedure and pass to the sort procedure the array of integers to be sorted. Figure 1.17 shows a sequence of statements.

```
=====
SORT(XARRAY);
=====
```

#### PARAMETER PASSING - FIGURE 1.17

Statement "SORT (XARRAY);" calls the sort procedure and passes to it the values XARRAY to be sorted. Often, we pass many parameters to a procedure. Three mechanisms exist for making parameters available to a procedure which is called. These are: 1) use of Global variables; 2) passing parameters by "value"; and 3) passing parameters by "reference".

The Global variable is the simplest mechanism for parameter passing. Of course, Procedure TEST can re-assign (or destroy) values in the variable "X". This means that "X" is not very secure. It is not well shielded. It is shared. So Procedure TEST can pass information back to the main program through variable "X" or any other global variable.

A second method of passing parameters is by value. If the parameter XARRAY, in figure 1.17, were passed by value, a copy of the value(s) in XARRAY would be made for the procedure SORT. In this way, SORT would have access to all the values of XARRAY, but would not be able to alter any of the original XARRAY values. You can see that this would provide a lot of protection for our original values in XARRAY. Hence, shielding or protection is excellent. However, communication is poor. The procedure SORT would not be able to return the sorted values in the original array. Some other mechanism would have to be used. Perhaps this would not be the best sort procedure.

A third method of parameter passing is by reference. If the parameter XARRAY in figure 1.17 were passed by reference, a pointer to the array XARRAY would be passed to the procedure SORT. This pointer would point to the original XARRAY. Executable statements in procedure SORT would then be able to reference and modify the original values in XARRAY. This is probably the best way for a SORT procedure. Parameters passed by reference are not especially secure (or protected) but enjoy the ability of providing two way communication.

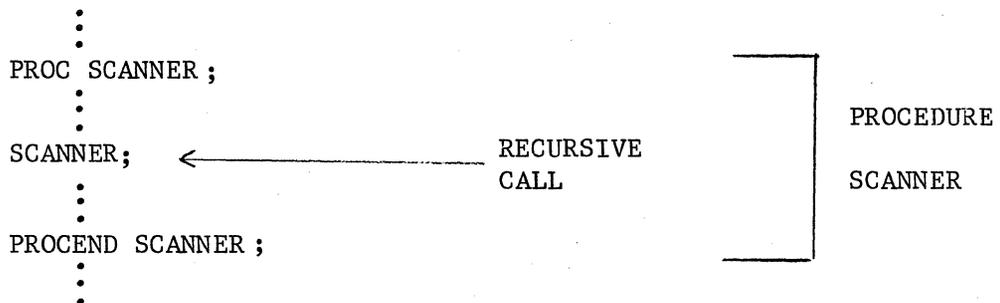
The chart on figure 1.18 provides some insights into the various uses of the three calling mechanisms.

PARAMETER PASSING METHOD	PROTECTION (SHIELDING)	COMMUNICATION (SHARING)
GLOBAL VARIABLES	WORST	BEST
CALL BY VALUE	GOOD	POOR
CALL BY REFERENCE	POOR	GOOD

PARAMETER PASSING MECHANISMS - FIGURE 1.18

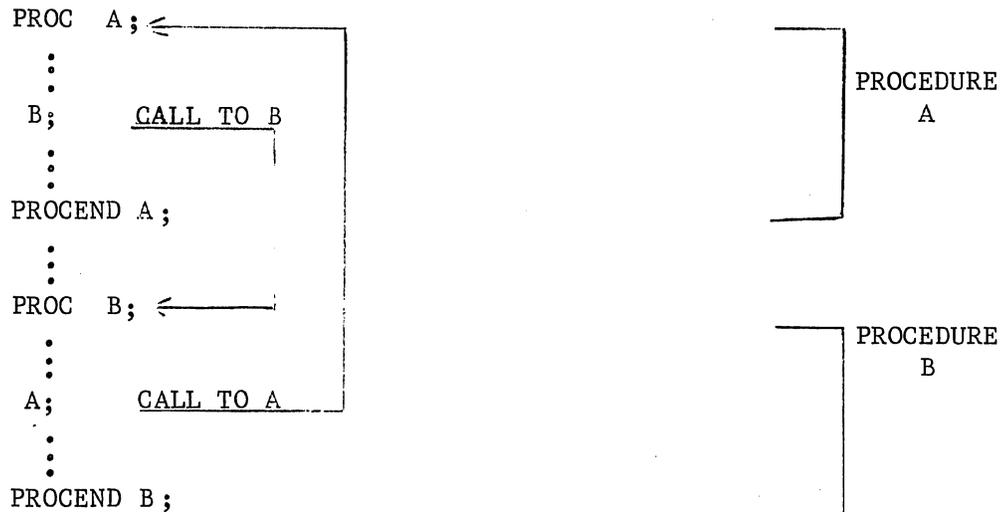
As you can see from the information presented in figure 1.18, no single parameter passing method provides the best all-round combination of protection and communication of variables. What we attempt to do is pick the right combination of protection and communication for each individual procedure.

Once procedures are written, we know that they are called by simply using the name of the procedure as a statement (on a line). When a procedure calls itself we say that the procedure is being used Recursively. For instance, figure 1.19 shows a very simple example of recursion.



SIMPLE RECURSION - FIGURE 1.19

We could imagine more complex examples of recursion. For example, suppose that a statement in Procedure A calls Procedure B and that a statement in Procedure B calls Procedure A. Recursion occurs here also, but the chain of procedure calls is longer. This example is illustrated in figure 1.20.



NOT-SO-SIMPLE RECURSION - FIGURE 1.20

All we need say here is that these recursive calls are allowed. Some programming algorithms are expressed very concisely using recursive techniques. Since ISWL supports recursive procedures we can implement these recursive algorithms in a straightforward manner.

Functions are a special kind of procedure. Like procedures, functions are executable statements which are called from some other statement. Functions follow all the rules of block structure and scope of variables. Functions may be passed parameters exactly as procedures (global variables, call by value, or call by reference).

Functions differ from procedures in two important ways. 1) the method of calling the function is unique, and 2) the function has a unique way (in addition to the conventional manner discussed for procedures) of returning values.

A function call cannot stand as a statement by itself. The function call must be part of another statement. For example, consider a function to return the square root of its real argument as shown in figure 1.21.

```

:
:
X:= 5*SQRT (Y);
:
:

```

FUNCTION CALL - FIGURE 1.21

In figure 1.21, we have an assignment statement that assigns X a value 5 times the square-root of Y. Of course Y must be assigned a value and the Function SQRT must be defined somewhere.

What is important, however, is that the call to SQRT is imbedded in statement.

The function value (square-root of Y) is returned where the function is invoked. In figure 1.21, the square root of Y is returned and is then multiplied by 5 to obtain a result which is assigned to X.

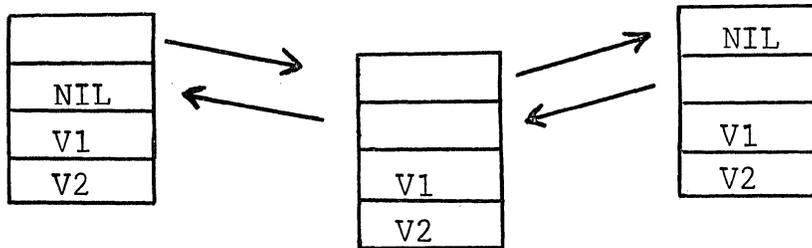
## POINTERS

A pointer variable is a variable whose value represents the location of some other variable (or ISWL element). Pointers are used effectively to:

1. Maintain the location of (or point to) elements in stacks, sequences and heaps (see page 1-19).
2. Point to procedures or labels.
3. Point to user declared elements, (i.e. arrays, records, variables, etc.).

There are, of course, other uses for pointers but these are the major uses.

One example of using pointers is illustrated in figure 1.22.



A FORWARD & BACKWARD LINKED LIST - FIGURE 1.22

Figure 1.22 illustrates a forward and backward linked list. Each list element (known as a record) contains two values (V1 & V2) and two pointers. The first pointer points to the next element in the linked list and is called the forward pointer (or forward link). The second pointer in each record points to the preceding record and is therefore called the backward pointer (or backward link). NIL is the value we give to a pointer that doesn't point to any element.

The beginning of the LINKED LIST may be identified because it has no backward pointer (no record precedes it). The end of the LINKED LIST may be identified because it has no Forward pointer (its forward pointer is NIL indicating that no list element comes after this one).

Another example of the use of pointers is a "Jump Table". The jump table is simply a table of pointers that usually point to procedures.

1	Pointer to Procedure X1
2	Pointer to Procedure X2
3	Pointer to Procedure X3
4	Pointer to Procedure X4
5	Pointer to Procedure X5

JUMP TABLE - FIGURE 1.23

Figure 1.23 illustrates the construction of a jump table. In this table there are five elements (or five entries in the table). Each entry is a pointer to some procedure.

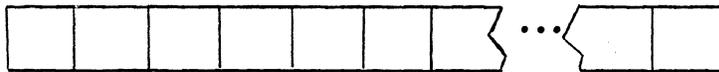
For instance, the table in figure 1.23 might be used to process Channel interrupts. If an interrupt occurs on Channel 4 we would look in the fourth entry in our jump table and the pointer (to Procedure X4) could be used to call procedure X4 to process this particular kind of interrupt.

There are many other uses for pointers and many will emerge later in the text.

## STORAGE MANAGEMENT

When we speak of storage management we are considering stacks, sequences and heaps. All of these concepts involve the use of storage. Often, we need special methods of accessing storage. These differing methods are embodied in our storage management capabilities.

Perhaps the easiest storage management scheme is the sequence. The sequence allows only sequential access to its member elements. A programmer will choose the sequence when he requires sequential access to data.

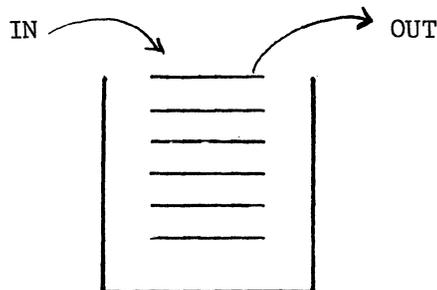


THE SEQUENCE - FIGURE 1.24

Figure 1.24 illustrates a sequence. The elements of a sequence are programmer defined. Each element could be an integer, a string, an array, a record, etc. Each sequence has a pointer associated with it. This pointer may be RESET to the beginning of the sequence (with a RESET statement). The pointer to the sequence may be advanced to the NEXT element in the sequence (with the NEXT statement). At any time, the programmer may reference the data in the sequence that the pointer points to. When the end of the sequence is reached (by successive applications of the NEXT statement), the sequence pointer becomes NIL (i.e. there are no more elements in the sequence).

You can see that the access mechanism for a sequence is clearly sequential.

The stack is simply another method of using storage. The stack is sometimes called a push-down stack to emphasize the method of accessing data in the stack. The elements of a stack are accessed using the LAST IN-FIRST OUT METHOD.

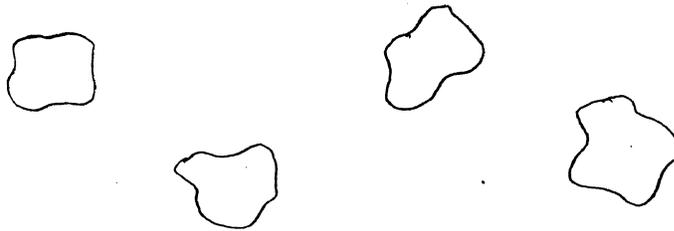


THE STACK - FIGURE 1.25

Figure 1.25 illustrates the concept that when elements are added to the stack (with a PUSH statement) they are placed "on the top of the stack". When elements are removed from the stack (with the POP statement) they are removed "from the top of the stack". This then is the way the LAST-IN, FIRST-OUT access is implemented. Of course, a stack can be "emptied" with the RESET statement.

The programmer will choose to use the stack when the FIRST IN - LAST OUT access method is the correct method for the problem at hand.

The last storage management method is the Heap.



THE HEAP - FIGURE 1.26

Figure 1.26 illustrates that the heap does not have an explicit access mechanism. Elements are placed in the Heap with the ALLOCATE statement. When the ALLOCATE statement is used, a pointer is returned to provide the ability to access this element in the future. The programmer must save the pointer returned. Additional elements can be placed in the Heap with additional ALLOCATE statements.

When the programmer no longer needs some (previously allocated) space he/she may "unuse" the space with the FREE statement. Each FREE statement returns (or makes available for future use) one element in the Heap. How do we know which element is FREE'd? The FREE statement must contain a pointer to the element in the Heap that is to be FREE'd.

In summary then, the different storage management statements simply provide the programmer with easy-to-use methods for accessing data. These methods conform to the accessing methods we find most prevalent in systems programming.

## TYPE

In many programming languages some type is associated with various elements such as variables. For example, it might be said that a variable is type "integer". In SWL a mechanism is provided to separate the definition of a type (e.g., integer) from the declaration of variables. Of course, when variables are declared their type must be specified. But the key point is that the programmer may construct some type and give that type a unique identifier. The unique identifier, then, represents the programmer defined type.

In any language there are many so-called pre-defined types. In SWL the pre-defined types include INTEGER, CHARACTER (abbreviated CHAR), REAL, and BOOLEAN. The programmer may, in addition, define additional types including ORDINAL, SUBRANGE, POINTER, and STRUCTURED types.

We need not discuss all the types here. The significant concept is that the definition of a type may be separate from the declaration of variables. When a variable is declared in terms of some type then the variable is restricted to values denoted by the type and in addition some referencing notation may be implied.

```
TYPE
    TABLE = ARRAY [1..10] OF INTEGER;
```

### TYPE DEFINITION - FIGURE 1.27

In figure 1.27, TABLE is defined to be a type which is a ten element array of integers. The ten elements must be referenced 1 through 10. Note that no storage space is consumed by the type definition. The definition simply provides an identifier (TABLE) for a type which is an array as described above.

Later in the program, if a variable is declared to be of type TABLE, that variable will be allocated enough storage space to contain ten integers. In addition, the array elements can only be accessed as elements 1 through 10 and the contents of the array must be integers.

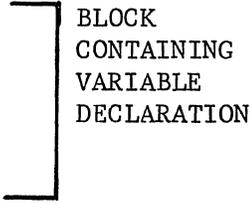
It would be nice if our programming language could check (at compile time and execute time) that all our assignments and computations do involve similar types. For example, if a pointer is declared to point to a REAL it would be nice if the programming language could "catch" the error when we tried to make that pointer point to an array. SWL provides a substantial amount of "TYPE CHECKING" at compile time and at execute time. In simple situations "TYPE CHECKING" seems like a nuisance. For instance, you cannot add an integer to a real (directly). First you must convert the real to integer and add two integers (or convert the integer to real and add two reals). The point is that the type conversion must be explicitly written. In complicated situations, the "TYPE CHECKING" done by the compiler (and at execute time) saves hours of difficult and tedious debugging.

VARIABLES

All variables used in a SWL program must be declared. When we declare a variable we specify the name or identifier of the variable, the type of the variable and other information such as access, storage and scope attributes.

A typical variable declaration is shown in figure 1.28

```
BEGIN
  :
  :
  VAR  X: INTEGER,
        Y: 0..15,
        R: REAL;
  :
  :
  END
```



VARIABLE DECLARATION - FIGURE 1.28

Figure 1.28 illustrates that the variable X is type integer, R is type real, and Y is a subrange of the integers. That is variable Y may be assigned only integer values in the range 0 to 15.

Variables declared as shown in figure 1.28 are valid (can be referenced) anywhere in the block in which they are declared. For a more detailed discussion of SCOPE of variables, see the section on Block Structure.

In addition, variables declared as in figure 1.28 are called, "Automatic Variables". This means that storage for these variables is created (or made available) when the block in which the variable is defined is invoked at execution time. When (at execution time) the end of the block is reached the storage space for these automatic variables is made available for other automatic variables. Hence, the lifetime of automatic variables is determined by the block containing the variable declaration.

It is possible to create a variable which "stays around" throughout the execution of the entire program. These variables are called static variables and their declaration is shown in figure 1.29

```
  :
  :
  BEGIN
  VAR  X: [STATIC] INTEGER,
        Y: [STATIC] 0..15,
        R: [STATIC] REAL;
  :
  :
  END
```

STATIC VARIABLE - FIGURE 1.29

In figure 1.29 the variables are declared to be static in addition to the other information which was discussed with figure 1.28.

The `STATIC` attribute causes storage to be allocated once and only once for each variable. This storage space once allocated will not be made available for other variables. The result is that the `STATIC` variable has a lifetime that includes the entire program execution.

There are other attributes for variables which will be discussed later in the text.

## UNDERSTANDING BNF

BACKUS - NAUR FORM (sometimes called BACKUS - NORMAL FORM) is used to describe the syntax of SWL. The syntax simply specifies VALID language constructs. The symantics or meaning of the language constructs are described in this text and in the SWL Reference Manual. Appendix B contains the BNF description of the SWL Language Syntax.

The specification of syntactic constructs are denoted by descriptions enclosed in the angle brackets  $\langle$  and  $\rangle$ . These words describe the nature or meaning of the construct. Constructs not enclosed in angle brackets stand for themselves. The symbol  $::=$  is read "is defined as" and the vertical bar  $|$  is used to denote alternative definitions and is read "OR". An optional syntactic unit (zero or one occurrence) is designated by square brackets  $[$  and  $]$ . Indefinite repetition (zero or more occurrences) is designated by braces  $\{$  and  $\}$ .

For example, consider the sample SWL BNF below:

```
 $\langle$  integer  $\rangle ::= \langle$  digit  $\rangle \{ \langle$  digit  $\rangle \}$ 
                |  $\langle$  digit  $\rangle \{ \langle$  hex digit  $\rangle \} \langle$  base designator  $\rangle$ 

 $\langle$  digit  $\rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ 

 $\langle$  hex digit  $\rangle ::= A | B | C | D | E | F |$ 
                a | b | c | d | e | f |
                 $\langle$  digit  $\rangle$ 

 $\langle$  base designator  $\rangle ::= (\langle$  radix  $\rangle)$ 

 $\langle$  radix  $\rangle ::= 2 | 4 | 8 | 10 | 16$ 
```

EXAMPLE OF BNF - FIGURE 1.30

In figure 1.30, we see that an  $\langle$ integer $\rangle$  is defined as a  $\langle$ digit $\rangle$  followed zero or more occurrences of a  $\langle$ digit $\rangle$ . And, a  $\langle$ digit $\rangle$  is defined as 0 thru 9. So a valid  $\langle$ integer $\rangle$  could be 6 or 637, etc. Also, we see the alternative definition of a  $\langle$ digit $\rangle$  is a  $\langle$ digit $\rangle$  followed by zero or more  $\langle$ hex digit $\rangle$  followed by a  $\langle$ base designator $\rangle$ . We can see what constitutes a valid  $\langle$ hex digit $\rangle$  and what is a valid  $\langle$ base designator $\rangle$ . Some examples of valid and invalid integer follow.

### INTEGERS

<u>VALID</u>	<u>INVALID</u>
7	FFF(16)
673	142(5)
16215(10)	
415(8)	
10110(2)	
OAEF(16)	

BNF INTEGERS - FIGURE 1.31

The valid integers do not require an explanation. The invalid integers are described below. FFF(16) is an invalid integer because according to the BNF an integer must begin with a digit and a digit is defined as 0 to 9. The proper way to write this value then would be OFFF(16). The integer 142(5) is invalid because the radix (or base) of five is not allowed. According to the BNF the only valid <RADIX> are 2, 4, 8, 10, and 16. If we really wanted to represent 142 Base 5, we could figure out its equivalent value in one of the correct Bases. For instance,  $142(5) = 2F(16) = 47(10) = 57(8) = 233(4) = 101111(2)$ . So we could use any of the valid representations shown above for this value.

## CHAPTER 2

### ELEMENTARY SWL

This chapter is a tutorial on the more elementary SWL language elements. Emphasis is placed on learning how to write simple SWL programs. Many examples are provided to clarify the material presented.

The following language elements are covered: basic constructs, symbols, identifiers, constants, variables, types (integer, real, character, boolean, subrange), declaration statements (MODULE, MODEND, VAR, CONST), assignment statements, structured statements (IF, LOOP, WHILE, REPEAT, FOR, CASE), control statements (EXIT), elementary input/output (READ and WRITE), and PROC [XDCL] MAIN;.

In every language, the programmer is faced with the problem of learning the symbols and keywords of the language, as well as, the rules for forming identifiers.

SWL provides a number of keywords which may be used only in special contexts. The programmer may construct identifiers for his/her use but programmer declared identifiers may not be the same as keywords. A list of keywords (or reserved words) is given in Appendix A.

Identifiers may be declared by the programmers. Identifiers are names containing 31 characters or less and begin with an alphabetic character. Upper and lower case letters are considered to be identical. Hence, an identifier written entirely in upper case letters is the same as the identifier written entirely in lower case letters. The valid characters after the first character (which must be alphabetic A to Z or a to z) include: the digits 0 to 9, the letters A to Z, the letters a to z, the underscore, the pound (or number) sign, the dollar sign, and the at sign.

Some examples of valid and invalid identifiers are shown in figure 2.1

<u>VALID</u>	<u>INVALID</u>
XYZ	\$VAR X
WHEAT_PRODUCTION	SIX+SEVEN
A&10	@loc
Syntax Table	4P3S7
SyStEm#VaLuE	_PETE_
FUNNY@VARIABLE	D.15
X#_\$_@	I/O
name_field	
X3P7S4	
Joe_	

IDENTIFIERS - FIGURE 2.1

Now that we have the basics of identifiers and we have referred to Appendix A and seen the list of keywords, lets put together the skeleton of a SWL program.

Every SWL program must be bounded by the MODULE and MODEND statements.

<u>ACCEPTABLE</u>	<u>PREFERRED</u>
MODULE;	MODULE INTRODUCTION;
:	:
:	:
:	:
MODEND;	MODEND INTRODUCTION;

MODULE & MODEND STATEMENTS - FIGURE 2.2

Figure 2.2 illustrates the beginning and end respectively of a SWL program. In the acceptable example, we have used the minimum information necessary (MODULE & MODEND) to describe a SWL program. The preferred approach shows the use of an optional identifier (in this case the identifier INTRODUCTION) to provide a name for the program (or module). It is generally considered good programming practice to give names to modules as these names enhance the readability of the source text. In the preferred approach then, we would say that the module is named (or called) "INTRODUCTION".

You will also notice the semi-colon (;) which is included in the example. In SWL, the semi-colon is used as a statement separator. A semi-colon is needed to separate one statement from another. There are places where a semi-colon may be omitted. In general, when a statement is followed by a keyword the semi-colon is not needed. Extra semi-colons may in general be used and will indicate an empty statement. Since the SWL compiler will ignore empty statements, extra semi-colons will not have any direct effect on the source program (i.e., they will not create errors). In some cases, the judicious use of semi-colons can make program modification easier, as will be shown later.

When writing source lines, one is apt to ask, "where can blanks go?". Generally, identifiers, reserved words, and constants must not contain imbedded blanks and must be separated, one from another by at least one blank.

Comments are added to the source text to enhance program clarity and may be used anywhere blanks can be used. The comment is enclosed in double quotes ("").

The comment itself may not contain a semi-colon {it looks like a statement separator to the compiler}, a double quote {it looks like another comment}, or a dollar sign {it is used to indicate a compile time option as described on page 4-28}.

```

MODULE COMMENT ILLUSTRATION;
"THIS MODULE ILLUSTRATES THE
USE OF THE COMMENT TO ENHANCE
PROGRAM CLARITY          "
:
:
MODEND COMMENT_ILLUSTRATION;

```

#### COMMENTS - FIGURE 2.3

In figure 2.3 the use of comments to enhance program clarity is illustrated. Figure 2.3 contains one three-line comment. It could have been written with three one-line comments.

All executable statements in SWL must be contained inside a procedure. We can cause the statements to be executed by calling the procedure. But what of the first procedure? How do we get that first procedure called? This is accomplished by the loader. After completion of the load, the loader simply calls the user procedure called MAIN. Since an external call is involved the procedure MAIN must be declared for external use. This is done with the XDCL (DeCLared eXternal) attribute.

So if we had some executable statements to perform (as any real program does) our Module(program) would look like the one shown in figure 2.4.

```

MODULE PROGRAM_STRUCTURE;

PROC [XDCL] MAIN;

"EXECUTABLE STATEMENTS"
"ARE PLACED HERE          "
PROCEND MAIN;

MODEND PROGRAM_STRUCTURE;

```

#### PROGRAM STRUCTURE - FIGURE 2.4

Notice in figure 2.4 above the procedure MAIN. The beginning of the procedure is denoted by "PROC [XDCL] MAIN;". PROC is a reserved word used to introduce a procedure. [XDCL] is used to indicate that the procedure is declared here and will be available for use by other external procedures (in this case the loader). MAIN is the name of the procedure. Of course, procedures can be given other names, but remember that the loader always transfers to the procedure named MAIN. The end of the procedure MAIN is indicated by the statement "PROCEND MAIN;".

The statements PROC to PROCEND define a procedure block. Inside this block we may have declarations for variables and executable statements. These variables and executable statements will follow the block-structuring rules discussed in Chapter 1.

Now lets go further and introduce some variables into our program. Variables are declared with the VAR statement. In the VAR statement, we provide (minimally) the name or identifier of the variable and the type of the variable. There is additional information that we can specify which we will discuss later. The simple variable declaration then, might appear as:

```
VAR X: INTEGER;
```

In this statement VAR indicates that we are declaring a variable and also introduces (or begins) the VAR statement. X is the identifier or name of the variable and INTEGER is the type of the variable. Notice the syntax. A colon is used to separate the variable identifier from the type. Since we have said no more, variable X is an automatic variable. That is, storage space for X will be allocated when the block in which the VAR statement appears is entered (at execute time). Since X is type integer, the permissible values that X may contain are restricted to the integers.

Often in a program we need many variables. The following statements illustrate how this may be done:

```
VAR X: INTEGER;  
VAR Y: REAL;  
VAR Z: BOOLEAN;  
VAR A: BOOLEAN;  
VAR C: CHAR;
```

Notice that we have declared five automatic variables: X, Y, Z, A, and C. Notice the use of five separate VAR statements. The semi-colon is used to separate the statements one from another. Variable Y is type REAL. That is, Y may be assigned any real value. Variable Z is declared to be type BOOLEAN. Variable Z, therefore, may contain one of the BOOLEAN values TRUE or FALSE. Variable C is type CHARACTER. Notice that we do not spell out the word character. We use instead the abbreviation CHAR. Variable C, then, may contain only a single character.

It seems like a waste of space to have to declare VAR five times, once for each variable. Wouldn't it be nice if we could use only a single VAR statement and declare lots of variables? Well, we can. Look below:

```
VAR X: INTEGER,  
    Y: REAL,  
    Z: BOOLEAN,  
    A: BOOLEAN,  
    C: CHAR;
```

Notice the interesting syntax here. VAR introduces the variable declarations. The semi-colon separates this statement from any following statements. Note that all five variables are declared in ONE statement. See the use of commas to separate one part of the variable declaration from another. This is a compound statement - but it is only one statement NOT five.

Now look at the variables Z and A above. They are both BOOLEAN. What a waste - having to write BOOLEAN twice. Can this be improved upon? Yes indeed:

```
VAR X: INTEGER,  
    Y: REAL,  
    Z,A: BOOLEAN,  
    C: CHAR;
```

In the statement above, when more than one identifier appears on the left of the colon (as in Z,A) the identifiers are separated, one from another, by a comma and they become the type declared to the right of the colon. In the example above, both Z and A are BOOLEAN.

How could we improve on this? Well, consider the problem of adding a variable to this declaration at the end (after C:CHAR;). What needs to be done? First we must change the semi-colon to a comma. Second we must add another line at the end. Two distinct lines must be changed. Similarly, if you consider adding a variable at the beginning of the VAR statement two lines must be altered. Wouldn't it be nice if we could add lines at the beginning or end and only have to make one change, the addition of the line? The following example illustrates the syntax necessary to make this possible:

```
VAR  
    X: INTEGER,  
    Y: REAL,  
    Z,A: BOOLEAN,  
    C: CHAR,  
    ;
```

We still have one statement, but now we can simply add new lines to this statement with the minimum amount of effort. This is just a technique. Unfortunately, this technique will not work in the current version of ISWL. The technique is mentioned here to increase your anticipation of future versions of ISWL.

Now we come to placing variable declarations in our Module. Figure 2.5 illustrates two alternatives:

<u>ALTERNATIVE 1</u>	<u>ALTERNATIVE 2</u>
MODULE VARIABLES1; VAR X: INTEGER;	MODULE VARIABLES2;
PROC [XDCL] MAIN; "PROGRAM BODY" PROCEND MAIN;	PROC [XDCL] MAIN; VAR X: INTEGER; "PROGRAM BODY" PROCEND MAIN;
MODEND VARIABLES1;	MODEND VARIABLES2;

#### SCOPE OF VARIABLES - FIGURE 2.5

Alternative 1 in figure 2.5 has the variable declaration inside the Module but outside any procedures (like MAIN). In this case, X is a global variable. Its scope (see Chapter 1) is the entire Module. Every procedure inside the Module (like MAIN) can make references to or alter the contents of variable X.

Alternative 2 in figure 2.5 illustrates placing the variable declaration inside a procedure (in this example, MAIN). With this approach, variable X is local to procedure (or block) MAIN. Variable X can be referenced or altered by any statement within procedure MAIN. However, if there were other procedures in this Module the other procedures would not have access to variable X (unless X was passed as a parameter to the procedure).

Generally, we place variable declarations in the block in which they are needed. This "shields" the variables from unintentional alteration by other procedures and makes for the most efficient allocation of storage for automatic variables.

Sometimes, when we declare a variable we don't want the variable to be able to take on all possible values normally associated with its type. For example, a variable declared as integer can take on all positive and negative integer values. It would be nice if we could restrict the values of some variable to the subrange 0 to 100 for instance. This can be accomplished with the use of "subrange". For example,

```
VAR INDEX : 0..100;
```

declares variable INDEX to be type "Subrange of INTEGER". We know the type is a subrange because of the use of ".." to indicate lower and upper bounds on the subrange. The type is subrange of integer because both 0 and 100 are integers. With this declaration the variable INDEX can be assigned (or take on) only integer values from 0 to 100 inclusive. Any other value placed into INDEX would be an error. We might also note that subranges are always expressed in ascending order (i.e. 100..0 is an illegal subrange).

Another example can be shown with the type character (CHAR). If a variable is declared to be type CHAR, then, it may contain any one of the 256 ASCII characters. But what if we really want a variable to be able to take on values equivalent to the alphabetic characters 'A' to 'Z'. Notice that characters are enclosed in single quotes (''). This is to distinguish the character 'A' from the variable identifier A. The following example shows the use of subrange of type character.

```
VAR ALPHA : 'A'..'Z';
```

In this example, the variable identifier ALPHA is a subrange of the characters. Since the lower bound ('A') of the subrange is a character, we can see that ALPHA will be a subrange of characters.

Is it possible to have a subrange of the type BOOLEAN? Type BOOLEAN only has two values FALSE and TRUE. So a subrange here would not provide us with any additional capabilities even though we could define such a subrange (i.e., VAR YES\_NO : FALSE..TRUE;).

What about subranges of the REAL type? Real values are written with an integer part, a decimal point and a fractional part (i.e., 1.0 is a real number). Also, the exponential form of real numbers is allowed (i.e., 1.3E6, represents  $1.3 \times 10^6$  or the value 1300000). Can we have subranges of these real values? Would it make sense to have a subrange 1.5..1.6? You might argue that this is alright, simply restricting a real variable to some specific subrange. In ISWL, however, subranges of the REAL type are not allowed.

What about the problem of numbers to bases other than base 10? In SWL the bases (or radix) 2, 4, 8, 10, and 16 are allowed. So, for example, to represent some binary subrange the programmer might write

```
VAR BINVAL : 0000(2)..1111(2);
```

to represent a variable whose binary values could be from 4 bits of zero to 4 bits all ones. Of course there are alternative ways of writing the same variable, as shown in figure 2.6.

```
VAR
  BINVAL1: 0000(2)..1111(2),
  BINVAL2: 0(4)..33(4),
  BINVAL3: 0(8)..17(8),
  BINVAL4: 0..15,
  BINVAL5: 0..OF(16);
```

#### SUBRANGES WITH RADIX - FIGURE 2.6

All the variables in figure 2.6 are automatic variables and they all have equivalent subranges. We have simply shown alternative ways of writing the subrange 0..15 (in decimal).

Occasionally, we need a constant in a program. It would be nice if we could give the constant a name (or identifier) and use the name instead of a constant. In addition, if the constant were needed in lots of places in the program we could use the identifier instead of writing the constant over and over again. This would have some added benefit if we had to change the constant at a later date. We could simply change the definition of the constant identifier in one place. All the references to the constant identifier would remain unchanged. Still another advantage is provided by the SWL compiler. When a constant is declared, it becomes a read-only constant. That is the constant cannot be changed by the executing program. Constants are declared with the CONST statement.

Just like VAR statement, the user can write a single CONST statement which declares many constants or many CONST statements each declaring constants. As an example, suppose that we have some lower and upper bounds for subranges. These could be declared as constants as shown below:

```
CONST
  LOWERBOUND = 0,
  UPPERBOUND = 100(16);
```

In this example notice the use of the equal sign (=). The equal sign is used in the CONST statement to imply an equality between the identifier on the left and the value on the right.

Naturally, we can declare INTEGER, REAL, CHAR and BOOLEAN constants. The example in figure 2.7 illustrates the use of constants.

```
MODULE CONST_EXAMPLE;
CONST
  LOWERBOUND = 0,
  UPPERBOUND = 100(16),
  CHARLIMIT  = 'Z',
  REALCONST  = -1.635,
  YES        = TRUE;

PROC [XDCL] MAIN;
VAR
  INDEX : LOWERBOUND..UPPERBOUND,
  ALPHA : 'A'..CHARLIMIT,
  RANGE : -100..LOWERBOUND;

"PROGRAM BODY"
PROCEND MAIN;

MODEND CONST_EXAMPLE;
```

#### CONSTANT DECLARATIONS - FIGURE 2.7

In figure 2.7 we can see the use of constants in variable declarations. Of course, constants may be used in other executable statements too. However, we have not covered any executable statements yet.

## INPUT/OUTPUT

For elementary Input and Output ISWL (not SWL) provides two statements: READ and WRITE. These statements are really procedure calls that cause reading of information from (or writing information to) a file. The READ and WRITE statements normally use the Input and Output files, respectively. However, we can alter this by parameters (on the control cards) to READ from (or write to) any arbitrary file.

What kinds of data can be read? The READ procedure is capable of reading integers, reals and characters. The WRITE procedure allows writing of integers, reals, characters, booleans and strings.

For example, the statements

```
VAR X:INTEGER;  
READ(X);
```

declare an integer X and then read an integer from the input file. The next integer on the file then becomes the value of X. When reading integers from a file, end-of-lines on the file are treated as blanks. Blanks are used to separate one integer from another. Each READ statement causes the next integer to be read from the file. Note that each read statement does not cause a new line to be read. Each read statement reads the next integer from the file.

In the following sequence

```
VAR CH: CHAR;  
READ(CH);
```

CH is declared to be type character. The read statement READ(CH) then reads the next character from the file.

In the case of reading characters, the end-of-line character is not treated as a blank. It is read as an end-of-line character. This enables us to distinguish one line from another.

A read statement may have many parameters as shown in the example below:

```
VAR  
  X: REAL,  
  C: CHAR,  
  Z: INTEGER;  
  
READ (X,Z,C);
```

The READ statement first reads a real number from the file. Then the next value on the file (which must be an integer) is read. Finally, the character following the integer is read.

The WRITE statement enables us to write information on the output file. The variables in the WRITE statement are added to the file in the order of their

appearance in the statement. Characters are written to the file in one character position. Integers are written to the file in 10 character positions. Reals are written in 20 character positions.

We can write end-of-lines on the file (to create new lines) by writing the constant EOL. For example, the statement

```
VAR I,J,K,L: INTEGER;
  ⋮
WRITE (I,J,EOL,K,L,EOL);
```

writes the value of I (right justified, blank fill) in 10 character positions on the output file followed by the value of J. Next, an EOL is written on the file followed by the value of K and L. The EOL will cause the values of K and L to appear on a line after the line containing I and J.

If we do not want the standard 10 character positions assigned to each real or integer written, we can specify the number of character positions to be used. For example, in the statement

```
VAR I : 0..20;
  ⋮
WRITE (I:5,EOL);
```

"I" is a subrange of the integers. "I" can never be less than 0 or greater than 20. The write statement specifies that the value of I (whatever it is) will be written on the output file using only 5 character positions. After the value of I, an end-of-line (EOL) will be written on the file.

For real values the WRITE statement may contain a specification like R:8:2.

```
VAR R:REAL;
  ⋮
WRITE (R:8:2,EOL);
```

In this WRITE statement, R:8:2 means write the real value R on the output file using 8 total character positions. The decimal portion of the real value R will contain 2 places.

Since we can write strings of characters on the output file we can put explanatory information on the output file. This is shown below:

```
VAR I : INTEGER;
  ⋮
WRITE ('RESULT = ', I:8,EOL);
```

The WRITE statement writes the string "RESULT =" on the file followed by the value of I (in 8 positions) followed by an end-of-line.

Now we can write a simple program that reads and writes data. Figure 2.8 shows a simple program.

```
MODULE INPUT_OUTPUT;  
  
PROC [XDCL] MAIN;  
VAR I: INTEGER;  
  
"READ AND WRITE AN INTEGER"  
READ(I);  
WRITE(I,EOL);  
  
PROCEND MAIN;  
MODEND INPUT_OUTPUT;
```

#### SIMPLE INPUT/OUTPUT - FIGURE 2.8

If we wanted to read and write two integers, we might choose either of the methods shown in figure 2.9.

MODULE IO1; PROC [XDCL] MAIN; VAR I,J : INTEGER;  READ (I,J); WRITE (I,J,EOL);  PROCEND MAIN; MODEND IO1;	MODULE IO2; PROC [XDCL] MAIN VAR I,J : INTEGER;  READ (I); READ (J); WRITE (I); WRITE (J,EOL);  PROCEND MAIN; MODEND IO1;
---	---

#### SIMPLE INPUT/OUTPUT (INTEGER) - FIGURE 2.9

In figure 2.9, we have two programs obtaining exactly the same results. These examples show that a READ list containing a number of variables is equivalent to a number of READ statements each containing a single variable.

Most often, however, we would like to do more than just read and write variables. We would like to perform some computations, make some assignments and write some results. So first let's examine the assignment statement.

## ASSIGNMENT

The assignment operator in SWL is ":=". The variable to the left of the colon-equal (:=) is assigned the value to the right of the colon-equal. Of course, the types must be conformable. For instance, if the variable on the left is type real and the resulting value on the right is type integer the assignment is not legal. Consider the following:

```
VAR X: INTEGER;  
  ⋮  
  X:=-3;
```

Variable X is declared to be type integer. The assignment "X:=-3;" is valid, because the type of the value (or constant) to the right of the assignment operator is the same as the type of the variable to the left of the assignment.

What about operators? What operators exist and how are they used?

## OPERATORS

SWL defines four classes of operators, these are:

```
NOT OPERATOR  
MULTIPLICATIVE OPERATORS  
ADDITIVE OPERATORS  
RELATIONAL OPERATORS
```

### CLASSES OF OPERATORS - FIGURE 2.10

The classes of operators are shown in figure 2.10. There is an implied precedence for these operators. The NOT operator has the highest precedence. Relational operators have the lowest precedence. Parenthesis are used to alter the normal order of precedence.

The multiplicative operators include multiplication (\*), division (/), remainder (MOD), and logical and (AND).

The additive operators include addition (+), subtraction (-), inclusive or (OR), and exclusive or (XOR).

The relational operators include less than (<), greater than (>), equal (=), greater than or equal (≥), less than or equal (≤), not equal (≠), set membership (IN) and a few other set operations.

When using these operators, the left and right arguments must conform. For instance, an integer may be added to an integer. A real may not be added to an integer.

In any expression, operations are performed in the order of precedence described above. In the following assignment statement

```
VAR I : INTEGER;  
  ⋮  
I:= 3*5+2;
```

the order of precedence is multiplication first and then addition; the result is I:=17. Using parenthesis we can alter the order of precedence as shown below.

```
VAR X: REAL;  
  ⋮  
X:= 3.0*(5.0+2.0);
```

Here multiplication should be done first. However, in order to do the multiplication the right argument "(5.0+2.0)" must be evaluated. The result (7.0) is then multiplied by 3.0 giving the result X:=21.0.

All this seems elementary, at best, until we encounter some interesting cases. These interesting cases can be shown using the relationals. Remember the relational  $A < B$  really asks a question: "is A less than B"? If the answer is yes the result is the Boolean value TRUE. If not, the result is the Boolean value FALSE. Try the next example:

```
VAR B: BOOLEAN,  
    I: 0..100,  
    Z: REAL;  
  ⋮  
I:=3; Z:=1.6;  
  ⋮  
B:= I<5 AND Z>1.5;
```

What do you think the value of B is? It appears that if I is less than 5 and Z is greater than 1.5 B will be true. Unfortunately, the AND operator has a higher precedence than the relationals. So the expression "5 and Z" is evaluated first which is invalid.

To fix the problem the expression must be rewritten with parenthesis as: "B:= (I<5) AND (Z>1.5);" so here the parenthesis are needed.

Now we can even do some elementary computation as shown in figure 2.10.

```
MODULE COMPUTE;  
PROC [XDCL] MAIN;  
VAR I: INTEGER;  
READ(I);  
I:=I*I*I;  
WRITE (I,EOL);  
PROCEND MAIN;  
MODEND COMPUTE;
```

ELEMENTARY COMPUTATION - FIGURE 2.11

The example in figure 2.11 reads an integer (I). I is then multiplied by itself three times (finds I to the third power) and writes out the result. In an expression like I:=I\*I\*I; where operators are of the same precedence the operations are performed left-to-right.

### CONVERSION FUNCTIONS

At this point you might ask, "what can we do if we really want to add an integer to a real"? This problem is addressed with the use of conversion functions. In SWL, conversion functions are constructed by preceding the type with a dollar sign. The argument to the conversion function then is placed in parenthesis. Some conversion functions are shown in figure 2.12.

<u>FUNCTION</u>	<u>ARGUMENT</u>	<u>RESULT</u>
\$REAL	INTEGER	REAL
\$INTEGER	REAL, CHAR	INTEGER
\$CHAR	INTEGER	CHAR

CONVERSION FUNCTIONS - FIGURE 2.12

For example, if we needed to add the integer I to some real value X and place the result in X the following method could be used:

```
VAR I: INTEGER,  
    X: REAL;  
:  
X:= X+$REAL(I);
```

In this example we have converted the integer I into a real value so that it could be added to X. The \$CHAR function is an interesting one. Remember that SWL uses the 256 ASCII characters. All the characters do not have graphic representations. How can we refer to a character we can't write (or type or punch)?

The answer is we convert an integer to the corresponding character. For example, \$CHAR (53) converts the integer 53 into the equivalent ASCII character. The result of \$CHAR (53) is a character. Remember the EOL character we discussed with respect to read and write statements? Well, \$CHAR(10) is the end-of-line character. If you are interested in the complete list of characters Appendix G shows the \$CHAR () equivalent of all the characters on the Control Data 713 terminal.

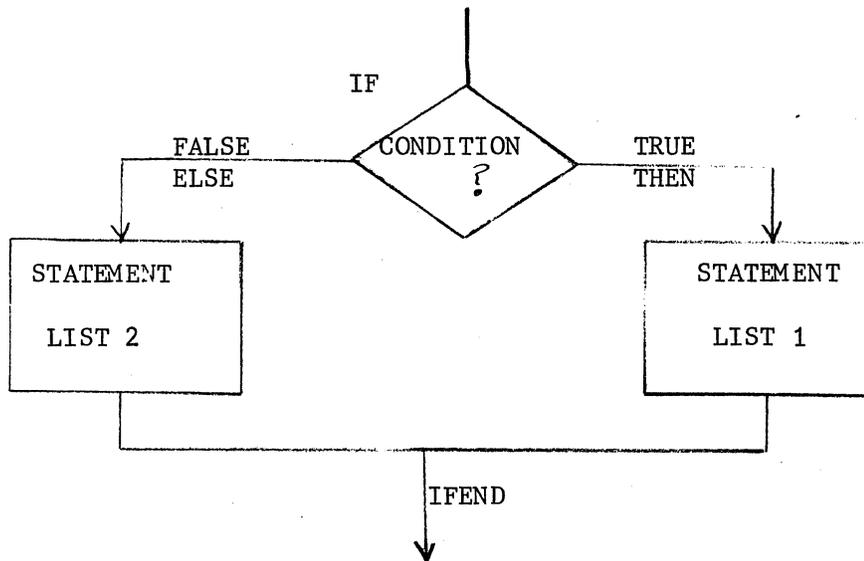
The integer conversion function can be used to convert a real into an integer or a character into an integer. For example, \$INTEGER('A') is the integer equivalent of the ASCII character 'A'. We might note in passing the identity \$INTEGER (\$CHAR(3)). This expression finds the character equivalent to the third ASCII character, and then finds the integer equivalent of the third ASCII character. The result is, of course, the integer 3.

### STRUCTURED STATEMENTS

Structured statements are statements that are composed of other statement lists. The SWL statements that fall into this category are: IF, WHILE, LOOP, REPEAT, FOR, and CASE statements.

### IF STATEMENT

The IF STATEMENT allows for the testing of conditions and for an alternative execution of statement lists.



IF STATEMENT FLOWCHART - FIGURE 2.13

Figure 2.13 illustrates the conditional test and the alternative statement list execution. The statement list may contain any SWL executable statement including additional IF statements. The beginning of the IF statement is denoted by IF and the end of the statement is denoted by IFEND. For instance, consider figure 2.13

```

VAR Y,X: INTEGER;
:
:
IF X<0
  THEN  Y:=X*X;
        WRITE (Y,EOL);
  ELSE  Y:=X/2;
        WRITE (Y,EOL);
IFEND;

```

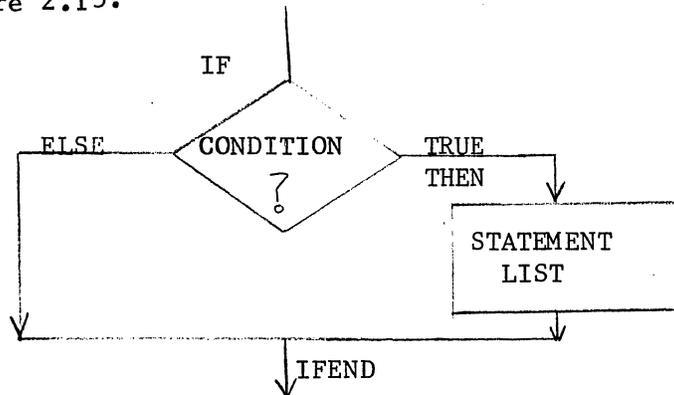
IF STATEMENT SYNTAX - FIGURE 2.14

In figure 2.14, the IF statement includes six lines. The one IF statement starts at IF and ends with the line IFEND. Notice that this statement includes other statements. This particular IF statement tests IF X is less than zero. If this is true, X<0 returns the Boolean result TRUE and the THEN part of the IF statement is executed. The THEN portion computes Y as X\*X and writes the resulting value of Y followed by an end-of-line. The next statement executed is the one after IFEND;.

In the event that X is not less than zero, X<0 is FALSE and the ELSE portion of the IF statement is executed. The ELSE portion computes Y as X divided by 2 and writes the resulting value of Y followed by end-of-line. The next statement executed is the one after the IFEND;.

So you can see how the SWL syntax implements the IF statement flowchart shown in figure 2.13.

Some variations on the basic IF statement are supported. One variation is when there is no ELSE portion. In this event, the ELSE clause may be left out entirely. The flowchart for this short form of the IF statement is shown in figure 2.15.



SHORT IF STATEMENT - FIGURE 2.15

As shown in figure 2.15, the short IF statement has only one statement LIST. An example of this form of the IF statement is shown in figure 2.16 below.

```

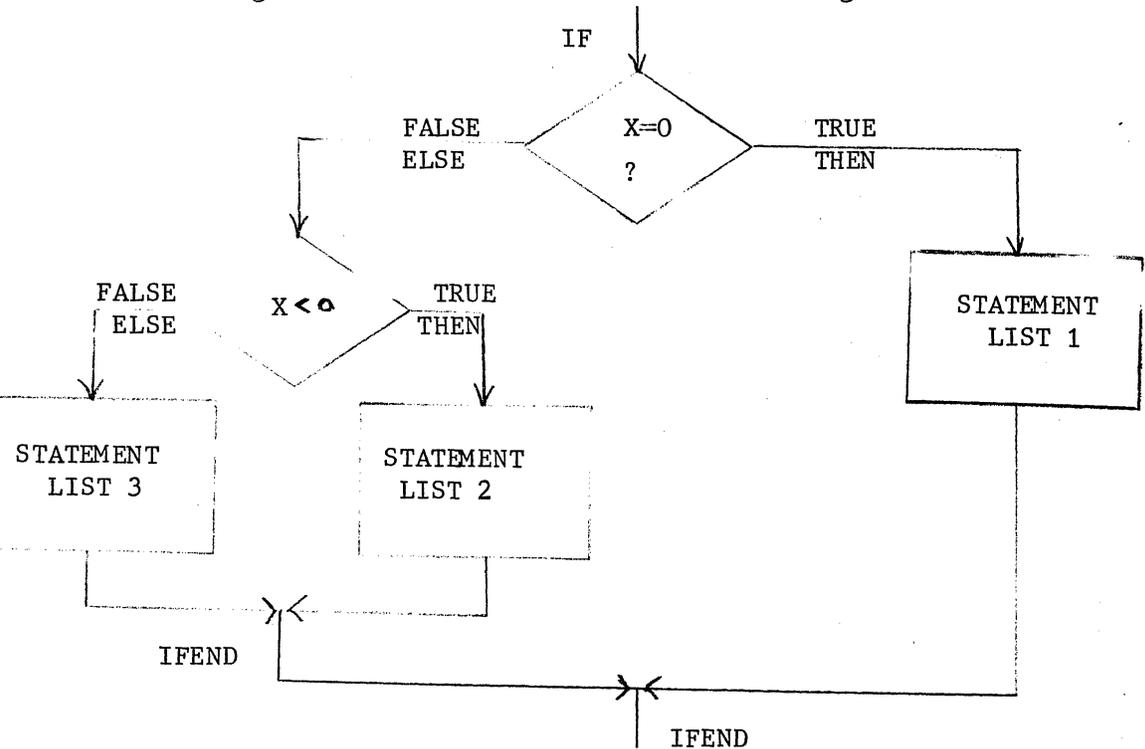
VAR DEBUG: BOOLEAN;
DEBUG := TRUE;
:
IF DEBUG
  THEN WRITE ('CHECKPOINT 17',EOL);
IFEND;

```

IF STATEMENT SYNTAX - FIGURE 2.16

Figure 2.16 is the kind of IF statement that could usefully be added to a program for checkout purposes. First, a BOOLEAN variable DEBUG is declared. If DEBUG is TRUE, the IF statement THEN clause will be executed writing some checkpoint information on the output file. If DEBUG is FALSE, the IF statement will not cause any additional output lines to be generated. That is, the IF statement effectively becomes a no-operation statement. We say "effectively" because some execution time is required to determine that DEBUG is FALSE and to skip over the executable statements in the IF statement. Also, the IF statement requires some space in the generated object code even if it never writes anything on the output file.

Sometimes we have many tests to perform in order to properly execute some statements. A fairly typical example is a three level test for less-than zero, equal-to-zero or greater-than-zero. This could be accomplished using nested IF statements as shown in figure 2.17.



NESTED IF STATEMENTS FLOWCHART - FIGURE 2.17

When we translate the flowchart of figure 2.17 into SWL SYNTAX it becomes somewhat complex as shown in figure 2.18.

```

VAR Y,X: INTEGER;
  ⋮
IF X=0
  THEN Y:=X*X;
  ELSE IF X<0
    THEN Y:=X+X;
    ELSE Y:=X-2;
  IFEND
IFEND;

```

NESTED IF STATEMENT SYNTAX - FIGURE 2.18

Since these nested IF statements can become quite complex SWL has introduced the ORIF clause which can (in many cases) simplify the complexity of nested IF statements. The ORIF clause provides the capability of performing additional tests within an IF statement without the necessity of creating another IF-THEN-ELSE-IFEND sequence. Figure 2.19 shows how the example of figure 2.18 can be simplified using the ORIF clause.

```

VAR Y,X : INTEGER;
  ⋮
IF X=0 THEN Y:=X*X;
ORIF X<0 THEN Y:=X+X;
ELSE Y:=X-2;
IFEND;

```

ORIF CLAUSE IN IF STATEMENT - FIGURE 2.19

Sometimes the tests in an IF statement are really individual cases. For instance:

```

VAR Y,X: 0..4;
  ⋮
IF X=0 THEN Y:= X*2;
ORIF X=1 THEN Y:= X+X;
ORIF X=2 THEN Y:= X-2;
ORIF X=3 THEN Y:= X-1;
ORIF X=4 THEN Y:= X/2;
IFEND

```

CASES IN AN IF STATEMENT - FIGURE 2.20

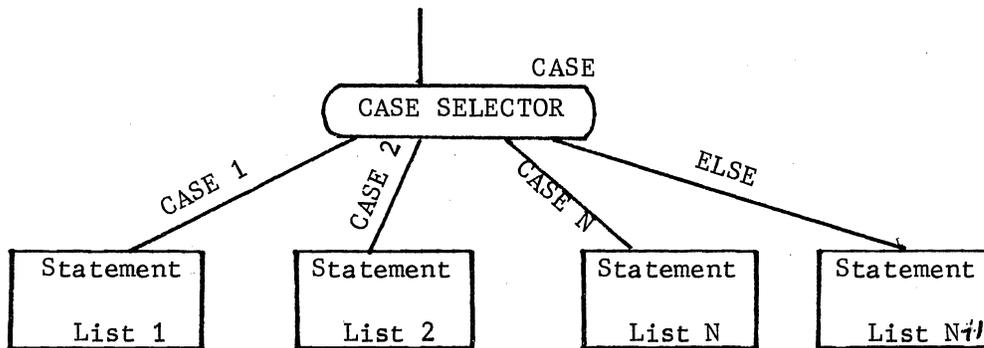
In figure 2.20, X can take on values of 0 thru 4, inclusive. The IF statement tests for which value X has and executes the appropriate

statement LIST. But when there are a finite (and relatively small) number of possible cases we have another statement which more clearly and succinctly explains the action at execute time. This statement is called the CASE statement.

### CASE STATEMENT

The CASE statement causes selection of one and only one of the constituent cases depending on the value of the case selector. The CASE statement can always be replaced by some equivalent IF statement(s). But, in many instances the CASE statement more clearly represents the intent of the programmer and hence improves program clarity.

The flowchart of the CASE statement is shown in figure 2.19.



CASE STATEMENT FLOWCHART - FIGURE 2.21

The ELSE portion of the CASE statement is optional. Let's take the example in figure 2.20 and show how it is made clearer with the use of the CASE statement as in figure 2.22.

```

VAR Y,X:0..4
:
CASE X OF
=0=  Y:=X*2;
=1=  Y:=X+X;
=2=  Y:=X-2;
=3=  Y:=X-1;
=4=  Y:=X/2;
CASEEND;
  
```

CASE STATEMENT SYNTAX - FIGURE 2.22

The CASE statement in figure 2.22 begins with the reserved word CASE and ends with CASEEND. This is one statement. The CASE statement is a structured statement, which can contain many other statements, but, it is still one statement.

In the CASE statement (in the first line "CASE X OF"), the variable X is called the selector and the values between the equal signs (like =0=) are called the cases. On entry to the case statement, the value of the selector is evaluated. This selector value is then used to select (or pick) the appropriate case for execution. For example, in figure 2.22 if X had the value 3 then the statement list "Y:=X-1;" would have been selected for execution. Upon completion of execution of this case, the statement after the CASEEND would be executed.

Another example of the CASE statement in figure 2.23 shows the use of the CASE statement to categorize characters in a syntactic analysis.

```

CONST
  ALPHA   = 1,
  NUMERIC = 2,
  SPECIAL = 3,
  OTHER   = 4;

VAR  CH : CHAR,
     CHTYPE : 1..4;

:
:
CASE CH OF
  ='A'..'Z'   = CHTYPE := ALPHA;
  ='0'..'9'   = CHTYPE := NUMERIC;
  ='+', '-', '/' = CHTYPE := SPECIAL;
  ELSE       CHTYPE := OTHER;
CASEEND;

```

CASE STATEMENT SYNTAX - FIGURE 2.23

In the example in figure 2.23 the CASE statement is used to classify a character. Notice the use of cases containing subranges. If the character CH is any of the characters 'A' to 'Z', then we set CHTYPE to ALPHA (which is the constant value 1. A similar action occurs if the character CH is a '0' thru '9'. Notice the use of non-contiguous cases. (='+', '-', '/'=). If the character CH is a plus, minus, or slash character, then CHTYPE becomes SPECIAL. The ELSE case is generally included as a good programming practice. In the event that none of the cases are selected, the ELSE case becomes effective.

Qh = How is a routine called :

Figure 2.24 illustrates the effective use of the ELSE case.

```
VAR X: 0..3,  
    Y: INTEGER;  
:  
CASE X OF  
=0=   Y:= X*X;  
=1=   Y:= X*2;  
=2=   Y:= X-1;  
=3=   Y:= X+1;  
ELSE  ERROR_ROUTINE;  
CASEND;
```

#### ELSE IN CASE STATEMENT - FIGURE 2.24

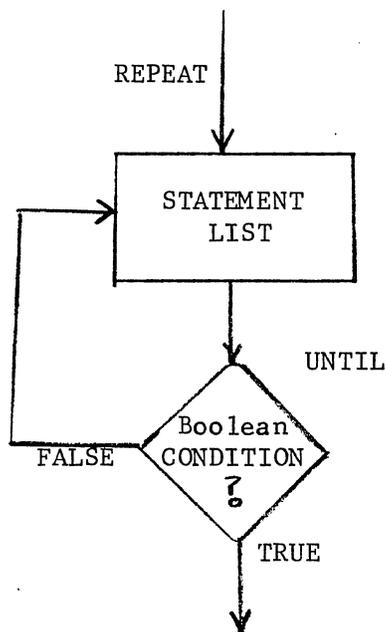
In figure 2.24 we can see that variable X is declared to be a subrange 0..3 of the integers. Certainly then, the CASE statement need only contain the cases =0=, =1=, =2=, and =3=. No other values for X are valid. If we could rely on this absolutely, the ELSE clause would not be needed. However, what happens when a memory parity error exists? What happens on some minor failure in the arithmetic unit of the computer? For these UNUSUAL conditions it is an extra safeguard to include the ELSE case. If X should ever erroneously obtain a value outside the proper subrange (0..3) and the case statement is entered the ELSE case would be executed calling a procedure "ERROR\_ROUTINE". Without the ELSE case and if X somehow becomes out of range the program would abort with the error message "CASE VARIABLE OUT OF RANGE". So, the ELSE case provides us, in this situation, an extra measure of program reliability.

#### REPETITIVE STATEMENTS

SWL provides a number of repetitive statements. These are: REPEAT, WHILE, FOR and LOOP. All these statements cause a list of statements to be executed repetitively. In some situations, the programmer could use any one of these statements and the choice of which statement to use will be an arbitrary one. However, each of these statements provides some special feature or capability not found in the other statements. These special features will be identified in the text below.

#### REPEAT STATEMENT

The REPEAT statement controls repetitive execution of its constituent statement list. The special feature of the REPEAT statement is that (upon entering the REPEAT statement) the statement list will be executed at least once. Thereafter, the boolean control expression will determine if additional repetitions are to be performed. The flowchart in figure 2.23 illustrates the flow of control in a REPEAT statement. Notice that the REPEAT statement begins with the word "REPEAT" and ends with the word "UNTIL". This is different than most other statements that end with some form of the word "END" (e.g. WHILEND, FOREND, ETC.).



REPEAT STATEMENT FLOWCHART - FIGURE 2.25

In the flowchart of figure 2.25, notice how the statement list is performed once. After the statement list is performed a boolean test is performed. If the result is FALSE, then the statement list is executed again. When the boolean condition result is TRUE then the repeat statement is finished and the next statement after the repeat is executed. Note that the statement list may contain any executable statements including additional (or nested) repeat statements. The repeat statement then can be read: "Repeat the statement list until some condition is true". Again, the special feature of this statement is that upon entry the statement list will be executed once regardless of the result of the boolean condition.

An example of the syntax of the repeat statement is shown in figure 2.26.

```

MODULE SKIP_LINE;
PROC [XDCL] MAIN;
VAR CH : CHAR
:
REPEAT
READ (CH);
UNTIL CH=EOL;
:
PROCEND MAIN;
MODEND SKIP_LINE;
  
```

REPEAT STATEMENT - FIGURE 2.26

The program in figure 2.26 will read characters from the input file (advancing the file pointer) until the end of line character is read. This effectively skips over (or goes past) the first line in the input file. It is assumed that at least one line is in the input file.

## Nesting of REPEAT Statement =

Notice that upon entering the REPEAT statement the "READ (CH);" statement will be executed. Then the test for end-of-line is performed. If this "READ (CH);" statement did not produce an end-of-line, then the statement list "(READ(CH));" is executed again.

What happens if the file is empty? That is, it contains only an end-of-file? The first read will read and set the end-of-file condition (no end of line is present). The conditional test (CH=EOL) will of course be false and the statement list (READ(CH)) will be executed again. This will cause an attempt to read past end-of-file and the job will terminate.

So, it would be nice if we could test for end-of-file, as well as end of line. Of course, we must execute the READ statement once in order to obtain any file information (end-of-line, end-of-file, or a character). The REPEAT statement guarantees that the READ statement will be executed once.

### END-OF-FILE

The end-of-file condition can be tested by using the boolean result returning built-in function #EOF(filename). This function can be used to test the status of any file. If end-of-file has been reached on the file, #EOF(filename) will be TRUE. If end-of-file has not been reached, then #EOF(filename) will be FALSE.

We can now modify the example in figure 2.26 to include the end-of-file test also. Naturally, once end-of-file is found some additional processing (not shown) will be required. Figure 2.27 illustrates the new (slightly) modified program.

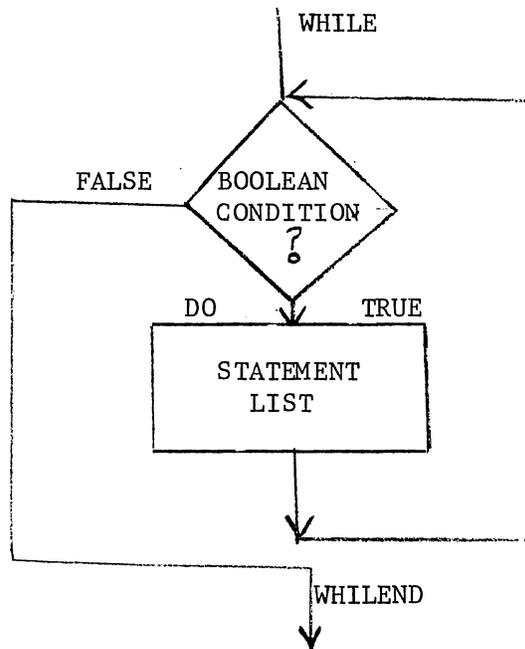
```
MODULE SKIP_LINE_NEW;  
PROC [XDCL] MAIN;  
VAR CH:CHAR;  
:  
REPEAT  
  READ(CH);  
UNTIL CH=EOL OR #EOF(INPUT);  
:  
PROCEND MAIN;  
MODEND SKIP_LINE_NEW;
```

REPEAT STATEMENT SYNTAX - FIGURE 2.27

### WHILE STATEMENT

The WHILE statement causes repetitive execution of its statement list but slightly differently than the repeat statement. The WHILE statement does the condition test first (upon entry to the while statement) and then executes the statement list only if the Boolean condition is TRUE. With this organization, it is possible that the statement list might not be executed at all.

The flowchart in figure 2.28 illustrates the operation of the WHILE statement.



WHILE STATEMENT FLOWCHART - FIGURE 2.28

An example of the use of the WHILE statement is shown below in figure 2.29. The idea here is to read an integer and compute the factorial of that integer, then terminate the program. We will restrict our input numbers to those that are 1) positive, including zero, and 2) less than 100. Remember 0 FACTORIAL is 1.

```

MODULE FACTORIAL;
PROC [XDCL] MAIN;
VAR N,FACT : INTEGER;
:
READ(N);
FACT:=1;

WHILE N>0 DO
FACT:= FACT*N;
N:= N-1;
WHILEND;

WRITE(N:5,'FACTORIAL = ',FACT,EOL);

PROCEND MAIN;
MODEND FACTORIAL;
  
```

WHILE STATEMENT - FIGURE 2.29

In the example in figure 2.29, notice how the WHILE statement performs the test for  $N > 0$  prior to the execution of any of the statement list. If the value 0 were read for N at the beginning of the program, FACT would be set to 1. Then the WHILE statement would evaluate  $N > 0$  and the result would be FALSE. In this event, none of the statements of the WHILE statement list would be executed. The WRITE statement then would write " 0 FACTORIAL = 1 " which is, of course, the correct answer.

After reading one value of N and computing one factorial, this program terminates. Let's modify the program slightly to continue reading values for N and computing factorials until a number greater than 100 appears on the input file. Since the input file may become empty (end-of-file), we will also include appropriate tests for this condition.

With these new features, we will have a reasonably complete and safe program. That is, the program will not be prone to aborting abnormally. See figure 2.30 below:

```

MODULE FACTORIAL1;
  PROC [XDCL] MAIN;
    VAR N,FACT : INTEGER;
    :
    READ(N);
    WHILE (N<=100) AND (NOT #EOF(INPUT)) DO
      FACT:=1;
      WHILE N>0 DO
        FACT:=FACT*N;
        N:=N-1;
      WHILEND;
      WRITE(N:5,'FACTORIAL = ',FACT,EOL);
      READ(N);
    WHILEND;

    WRITE ('END OF JOB',EOL);

  PROCEND MAIN;
MODEND FACTORIAL1;

```

WHILE STATEMENT (FACTORIALS) - FIGURE 2.30

Let's review the operation of the program in figure 2.30. The WHILE statements have been enclosed in brackets to illustrate where they begin and end, and also the nesting of the WHILE statements. Upon entry into the program a value of N is read. The WHILE statement then checks that 1) N is not greater than 100 and 2) that we have not encountered an end-of-file. If either of these conditions is false, then no statements in the constituent list of the WHILE statement are executed. The next statement executed will be the one that writes the end of job message.

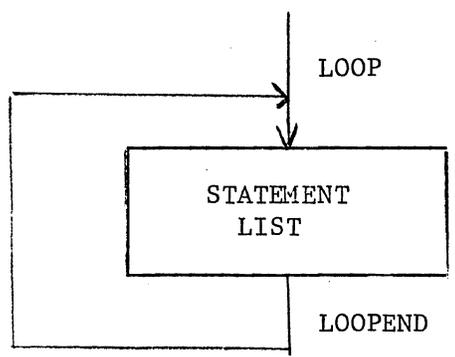
If N is within range and there is no end-of-file, then the WHILE statement is entered. The factorial is computed as described for the program in figure 2.29. The result is written to the output file and another N is read. Then control is returned to the beginning of the outermost while statement to check the conditions on N and end of file.

Could repeat statements have been used in the example in figure 2.30? The answer is of course yes - but, you will find if you write such a program that extra IF statements will be needed. This is because the repeat statement does its testing at the end of the statement list and in this example we need tests performed prior to executing the constituent statement list.

LOOP & EXIT STATEMENT

The LOOP statement provides for unbounded looping. The statements inside a LOOP statement are executed repetitively forever. The LOOP statement itself provides no mechanism for termination. Conceptually, then we might use a LOOP statement if we were writing some portion of an operating system, for example, which was always executing. The actual instances of this type of programming is relatively rare. So a mechanism for exiting (or getting out of) the LOOP statement is provided. This mechanism is called the EXIT statement.

The flowchart of a LOOP statement is very simple. It is shown in figure 2.31 below.



LOOP STATEMENT FLOWCHART - FIGURE 2.31

The EXIT statement can be used to exit any structured statement (IF, REPEAT, FOR, CASE, WHILE, LOOP).

The EXIT statement comes in a number of flavors. The simplest form is simply EXIT. When the EXIT statement is executed, it exits the structured statement in which it is enclosed. For example, in figure 2.32

```

IF A<B
  THEN EXIT
  ELSE A:=B*Z;
      WRITE (A,B,EOL);
IFEND;
  
```

EXIT STATEMENT - FIGURE 2.32

The IF statement in figure 2.32 contains an EXIT statement. In the event that A is less than B then the EXIT statement will cause control to be passed to the statement following the IFEND.

The EXIT statement in this example is not really needed. If the EXIT statement were omitted the THEN clause would be empty and the program would accomplish exactly the same result.

A more meaningful use of the EXIT statement is the following. An EXIT statement may contain a WHEN clause which will cause a conditional exit. This form of the EXIT statement is shown below:

```
EXIT WHEN A<B;
```

When the EXIT statement is executed and A is less than B, the EXIT statement will EXIT. If A is not less than B this statement becomes a no-operation. That is, it does not EXIT.

Now we can show how the LOOP statement and EXIT statements can be used together to provide a conditional repetitive statement. The example in figure 2.33 illustrates a method for copying a file.

```
MODULE COPY;  
PROC [XDCL] MAIN;  
VAR CH:CHAR;  
  
LOOP  
READ (CH);  
EXIT WHEN #EOF (INPUT);  
WRITE (CH);  
LOOPEND;  
  
PROCEND MAIN;  
MODEND COPY;
```

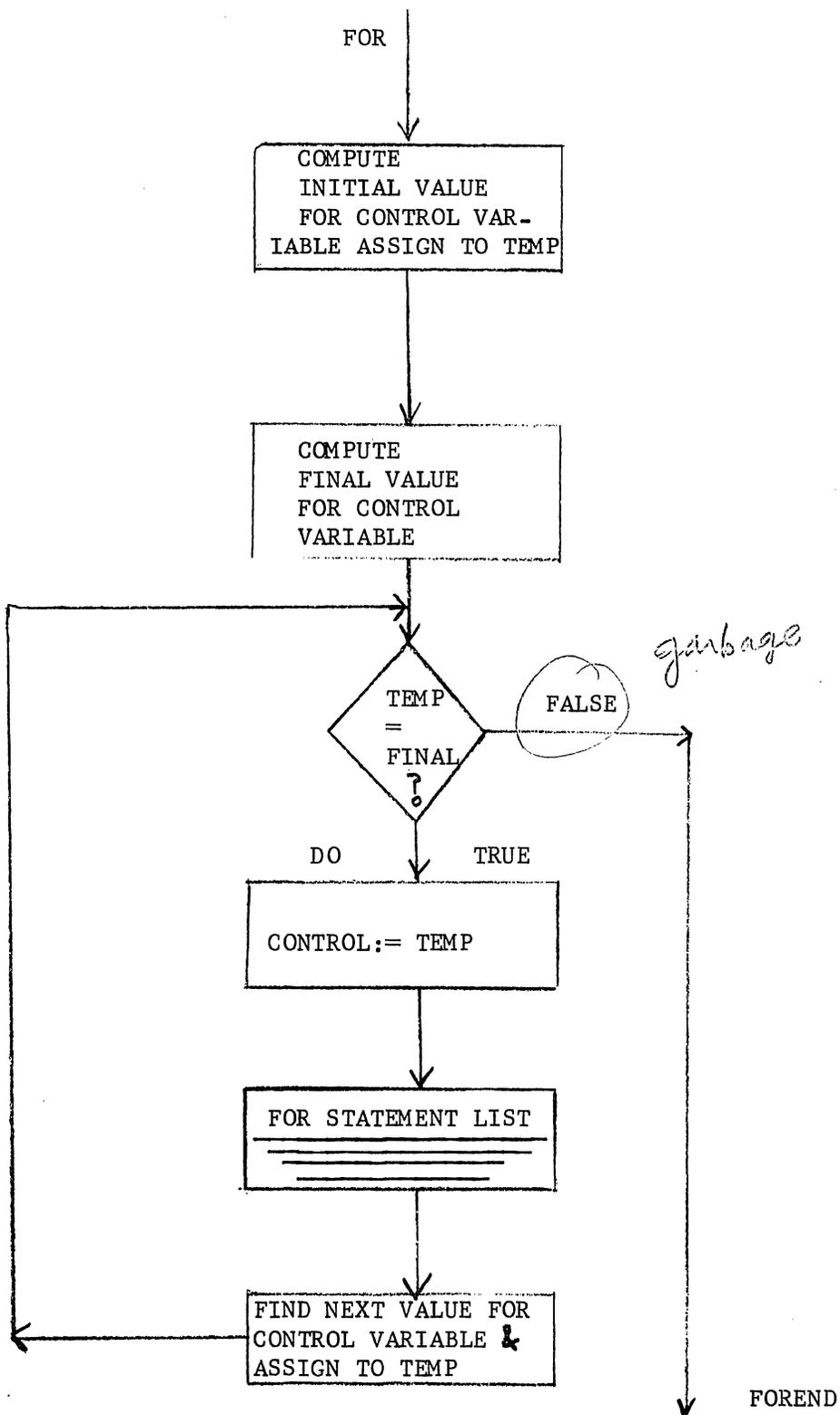
#### LOOP STATEMENT - FIGURE 2.33

The program shown in figure 2.33 reads characters from the input file and writes them on the output file. After each character read, a test is made for end-of-file. If the end of file condition is TRUE, then the loop statement is exited and of course the job terminates.

#### FOR STATEMENT

Our last repetitive statement is the FOR statement. This statement uses a control variable to control the execution of the constituent statement list. Control over the number of repetitions is declared explicitly rather than being some boolean test as is the case in the REPEAT and WHILE statements.

A flowchart of the FOR statement is shown in figure 2.34



FOR STATEMENT FLOWCHART - FIGURE 2.34

*I don't agree! may be ascending or descending.*

Figure 2.34 illustrates the FOR statement flowchart. This flowchart is really only valid for ascending FOR statements. Ascending FOR statements are those whose control variable takes on successively higher values. When the control variable takes on successively smaller values we say that the FOR statement is descending. The ISWL FOR statement can only increment or decrement by one.

An example of the FOR statement syntax is provided by the example in figure 2.35. In this example, assume that the input file contains a number of integer values to be added together. The first integer on the file, however, indicates how many additional integers are to be added together.

```
MODULE SUMS;
PROC [XDCL] MAIN;
VAR I,M,N,S: INTEGER;
S:=0;
READ(M);

FOR I:= 1 TO M DO
READ(N);
S:=S+N;
FOREND;

WRITE('SUM = ',S,EOL);
PROCEND MAIN;
MODEND SUMS;
```

#### FOR STATEMENT SYNTAX - FIGURE 2.35

Notice in the FOR statement in figure 2.35 that the control variable (I) will take on successive integers from 1 to M where M has been read from the input file. The FOR statement, then, is executed M times. Each time a new value is read it is added to the sum S. When the FOR statement is exhausted (after M iterations), the sum is written to the output file.

When we wish to choose control variable values in descending order we use the reserved word DOWNTO instead of TO. We might also note that the control variable need not be integer. The control variable can be any type whose successor and predecessor values may be found. For example, this would include the characters.

Have you ever had trouble saying the alphabet backwards? The example in figure 2.36 illustrates a program to produce the alphabet backwards so you will have the correct sequence to practice.

```

MODULE BACK_ALPHA;
PROC [XDCL] MAIN;
VAR I: 'A'..'Z';

FOR I:='Z' DOWNT0 'A' DO
WRITE (I,EOL);
FOREND;

PROCEND MAIN;
MODEND BACK_ALPHA;

```

FOR STATEMENT SYNTAX - FIGURE 2.36

In the example in figure 2.36 variable I is declared to be a subrange of the characters ('A' to 'Z'). That is to say that at execution time I may take on only alphabetic characters 'A' to 'Z' (but not necessarily in that order). The FOR statement indicates that I is to take on values from 'Z' DOWNT0 'A'. The WRITE statement writes each successive value of the control variable on a line by itself on the output file. In this way the characters Z Y X W etc. are listed on the output file.

#### REPETITIVE STATEMENT COMPARISON

While each of the repetitive statements (FOR, WHILE, REPEAT, and LOOP) has some special feature associated with it, there will be many times when the choice may be arbitrary.

As an example, lets look at the problem of creating some repetitive sequence from 1 to LIMIT. In our example, the value of the control variable must be available during each iteration.

How can we solve this programming problem with each of the four repetitive statements? The next four figures 2.37 thru 2.40 illustrate some solutions to the problem.

```

CONST LIMIT = 100;
VAR I: 1..LIMIT;
:
FOR I:= 1 TO LIMIT DO
==
==
==
FOREND;

```

REPETITIVE FOR STATEMENT - FIGURE 2.37

Notice VAR accepts arith. op.

```
CONST LIMIT = 100;  
VAR I : 1..LIMIT+1;  
:  
I:=1;  
REPEAT  
=  
=  
=  
I:=I+1  
UNTIL I > LIMIT;
```

REPETITIVE REPEAT STATEMENT - FIGURE 2.38

```
CONST LIMIT = 100;  
VAR I : 1..LIMIT+1;  
:  
I:=1;  
WHILE I <= LIMIT DO  
=  
=  
=  
I:=I+1;  
WHILEND;
```

REPETITIVE WHILE STATEMENT - FIGURE 2.39

```
CONST LIMIT = 100;  
VAR I : 1..LIMIT+1;  
:  
I:=1;  
LOOP  
=  
=  
=  
I:=I+1;  
EXIT WHEN I > LIMIT;  
  
LOOPEND;
```

REPETITIVE LOOP STATEMENT - FIGURE 2.40

You are bound to ask, "Which of the above four methods is the best?" And by that question you mean 1) which method requires the least execution time and 2) which method requires the smallest storage requirements or 3) which method is most easily understood and maintained? We don't know the answer yet - but we are working on it!

## CHAPTER 3

### SWL DATA STRUCTURES

This chapter presents both concepts and examples of structuring data in SWL. We begin with a discussion of SWL types. This includes both the declaration and use of types. We progress onto type conformity and the use of type testing. We will amplify the material on variables presented in the last chapter to include some of the more important variable attributes, data structures, and variable references. We will also discuss value constructors and variable initialization.

## TYPES

In the last chapter, we examined some of the more primitive (or elementary) types including integer, real, boolean, and character. The point was made that a clear distinction exists between the variable and the type of the variable. The simple variable declaration statements like "VAR X:INTEGER;" were used to declare an automatic variable X and its type-integer. Now, an expansion of the type specification facility is needed.

In SWL, the capability exists to declare a type. Not a variable - just the type. When this is done, the reserved word TYPE is used to introduce the declaration. A simple example is shown in figure 3.1.

```
MODULE TYPE_DECLARATION;  
  TYPE Q = INTEGER;  
  :  
  VAR X:Q;  
  :  
MODEND TYPE_DECLARATION;
```

TYPE DECLARATION - FIGURE 3.1

In figure 3.1 above, the reserved word TYPE is used to introduce the type declaration. Q is the name (or identifier) of the type. The equal sign is used to signify equality between the name Q and the type INTEGER. What has been accomplished is simply to provide an alternative identifier "Q" which stands for (or represents) type integer. You can see how this identifier can be used in the VAR statement. Instead of specifying "VAR X:INTEGER;" it is possible to declare "VAR X:Q;". Which simply says that X is to be type Q, but we know that Q is type integer.

So one thing an explicit type declaration allows is the ability to give a name to a type and use the name to stand for the type. This can be especially useful when the type itself is quite lengthy (say 5 or more lines, as in the case of records).

Another use for the explicit type declaration is that some SWL language elements make reference to type. An example is to declaration of pointers which must point to a type. In this case, the type declaration can be helpful.

In the last chapter, we examined the types integer, real, boolean, and character. Now we will examine some of the more complex (and useful) types.

## ORDINALS

Ordinals are used primarily to provide meaning and clarity to the program text. An ordinal is user defined and is scalar. That is the user defines the elements of the ordinal in some order. Once the order is declared it is fixed. In general, then, it is possible to determine the predecessor and successor for any element in the ordinal. Once ordinal identifiers are declared the identifiers may not be used in any other context in the program. The identifiers may be used solely to indicate the elements of the ordinal.

Let us assume that we have a program that processes grain production values. The grains we are interested in are wheat, oats, barley, and rye. We could define a type GRAIN which could be any of the four grains we are interested in, see figure 3.2.

```
TYPE
  GRAIN = (WHEAT, OATS, BARLEY, RYE);
```

#### ORDINAL TYPE - FIGURE 3.2

Notice the use of parenthesis in figure 3.2. Parenthesis around the four identifiers indicate that these are ordinal elements. We should also note that there exists a direct relationship between the ordinal identifier and its position. Wheat is the 0th ordinal element. Similarly, OATS is the 1st ordinal element, BARLEY is the 2nd and RYE the 3rd ordinal element.

Once an ordinal type is declared we can do lots of interesting things. For example, we can declare variables that may take on values of the ordinal as shown in figure 3.3.

```
TYPE
  GRAIN = (WHEAT, OATS, BARLEY, RYE);
:
VAR
  INDEX : GRAIN;
:
  INDEX := WHEAT;
```

#### ORDINALS - FIGURE 3.3

In figure 3.3 above a variable INDEX has been declared to be type GRAIN. That means that the variable INDEX can be assigned any of the appropriate "values" of the ordinal as illustrated by the "INDEX:=WHEAT;" statement.

How does all this improve program clarity? Assume that we wish to find the total production of grain when we know the production of wheat, oats, barley and rye. If the production values (4 of them) are real and exist on an input data file then the following program could get the task accomplished.

```
MODULE SUMMATION;
TYPE GRAIN = (WHEAT, OATS, BARLEY, RYE);
PROC [XDCL] MAIN;
VAR I : GRAIN
    V, PRODUCTION: REAL;
PRODUCTION:=0;
FOR I:=WHEAT TO RYE DO
  READ (V);
  PRODUCTION:= PRODUCTION +V;
FOREND;
WRITE ('PRODUCTION=', PRODUCTION, EOL);
PROCEND MAIN;
MODEND SUMMATION;
```

#### USE OF ORDINALS - FIGURE 3.4

Perhaps the most interesting aspect of the program in figure 3.4 is the statement "FOR I:= WHEAT TO RYE DO". Since WHEAT and RYE are unique identifiers (representing the 0th and 3rd elements of the ordinal GRAIN), we can use them as ordinal constants. Since the ordinal constants have a SCALAR ordering (successor and predecessor) the SWL compiler can generate code to find the successive values as required by the FOR statement. Note that the control variable in the FOR statement (I) must be type GRAIN in order to accept the values WHEAT..RYE. Isn't that program much clearer than one in which the FOR statement read "FOR I:= 0 to 3 DO"? If the FOR statement says "0 to 3" it is not clear what is being done. If the FOR statement says "WHEAT TO RYE" it is pretty obvious that the program is dealing with GRAIN.

*great feature*  
\*

As you might imagine these ordinal values can be used in subranges (WHEAT..OATS) and as cases in the CASE statement.

It was mentioned above that the ordinal identifiers are ordered and there is an equivalence (of sorts) between the 0th element in the ordinal list and the first identifier. How then can we convert an ordinal identifier into an integer and an integer into an ordinal identifier?

Figure 3.5 shows how an ordinal identifier can be converted into an integer using the \$INTEGER conversion function.

```
TYPE
  COLOR = (RED, YELLOW, BLUE, GREEN);
  VAR X: COLOR,
      I: 0..3;
:
I:= $INTEGER(BLUE);
```

#### CONVERTING ORDINALS TO INTEGERS - FIGURE 3.5

In figure 3.5 the statement "I:=\$INTEGER(BLUE);" converts the ordinal identifier BLUE into an integer (the value would be 2) and then assigns the value to an integer variable (I). The statement "I:=BLUE;" would not be valid since the type of I is integer and BLUE is a member of an ordinal. Our program must, therefore, make the types equivalent by using the conversion function \$INTEGER.

Now let's go the other way, converting an integer into an ordinal element as shown in figure 3.6.

```

TYPE
  HARDWARE = (NAILS, TACKS, SCREWS);
VAR
  H:HARDWARE,
  I: 0..2;
  .
  .
I:= 1;
  .
  .
H:= $HARDWARE (I);

```

#### CONVERTING INTEGERS TO ORDINALS - FIGURE 3.6

In figure 3.6 the statement "H:=\$HARDWARE (I);" takes the integer value I and converts it to type HARDWARE (in this case since I:=1, the conversion \$HARDWARE (I) will result in the ordinal TACKS) and the resulting ordinal, TACKS, would be assigned to the variable H.

It is often quite a convenience to be able to go back and forth between the ordinal identifiers and their equivalent integer representation.

#### ARRAYS

The array type is a convenient way of providing random access to a number of homogeneous elements. Arrays can have any number of dimensions (there is no arbitrary limit). An array has two significant components: 1) the indices and 2) the contents.

The general syntax of an array declaration is "ARRAY [indices] OF type". That is, in the declaration of an array the indices and type of components must be specified.

Perhaps the simplest array structure is one that is one dimensional, has integer indices, and contains integer components. This type would be declared as shown in figure 3.7.

```

TYPE
  TYPICAL = ARRAY [0..100] OF INTEGER;
VAR
  X:TYPICAL;

```

#### TYPICAL ARRAY TYPE - FIGURE 3.7

Figure 3.7 shows an array declaration that is typical. The array contains 101 distinct elements (0..100) and each element is an integer. The variable X is type TYPICAL which means that X is an array of 101 integer elements. In a program, the identifier X is used to refer to the entire array. To refer to any single element (integer) of the array the notation X[I] is used;

Range spec for control variables? like integer only?

"I" can be an integer from 0 to 100 (as defined by the TYPE declaration).

An example of the declaration and use of arrays is shown in figure 3.8.

```
MODULE ARRAY_SETUP;
PROC [XDCL] MAIN;
TYPE
  A = ARRAY [-5..5] OF INTEGER;
VAR
  X:A,
  I: INTEGER;

FOR I:= -5 TO 5 DO
X [I] := I*5;
FOREND;

PROCEND MAIN;
MODEND ARRAY_SETUP;
```

USE OF ARRAYS - FIGURE 3.8

In the example in figure 3.8, the variable X is type A. Type A is an array of integers. There are 11 elements (or indices) for this array -5 thru+5. The FOR statement selects successive values from -5 to +5 and assigns these to the control variable I. The array X is then initialized element by element.

We may also declare arrays with more than one dimension by declaring an array of an array. This may be done in either of two ways illustrated in figure 3.9.

```
TYPE
  TWOD1= ARRAY [1..10] OF ARRAY [1..10] OF REAL;

TYPE
  TWOD2= ARRAY [1..10,1..10] OF REAL;
```

MULTI-DIMENSIONED ARRAYS - FIGURE 3.9

Figure 3.9 shows the two ways for declaring a two dimensional array type. The first method is preferred as it clearly illustrates the concept of a two dimensional array as an array of an array. Regardless of which method is used to declare the array type, any variable that is a two dimensional array can be referenced in either of two ways, as shown in figure 3.10.

```

TYPE
A = ARRAY [1..10] OF ARRAY [20..30] OF INTEGER;
VAR
X:A;
.
X[5,21] := -67;
X [8][22] := 73;
.

```

ARRAY REFERENCES - FIGURE 3.10

Figure 3.10 illustrates the two (equally correct) methods of referencing a two dimensional array. The first method "X[5,21]" is correct but not preferred. "X[8][22]" is preferred because it clearly carries thru the idea of double subscripting and the idea of an array of an array. We should also note that using the example in figure 3.10, X[3] is a valid reference and refers to one array 20..30 of an integer. In fact each X[I] (where I is from 1 to 10) is an array 20..30 of integer.

Some examples of array concepts and their specification in SWL are given below:

?	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE

```

TYPE
TRUTHTABLE = ARRAY [BOOLEAN] OF
              ARRAY [BOOLEAN] OF BOOLEAN;

```

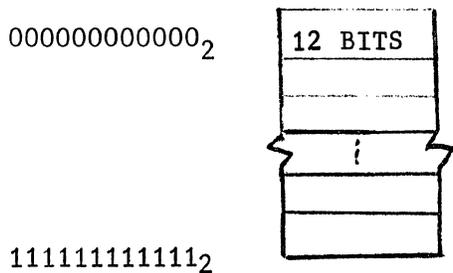
```

VAR
X: TRUTHTABLE;
.
X [TRUE] [TRUE] := FALSE;
X [TRUE] [FALSE] := FALSE;
X [FALSE, TRUE] := TRUE;
X [FALSE, FALSE] := TRUE;

```

} X [TRUE] == FALSE ;  
 } X [FALSE] == TRUE ;

A BINARY TRUTH TABLE - FIGURE 3.11

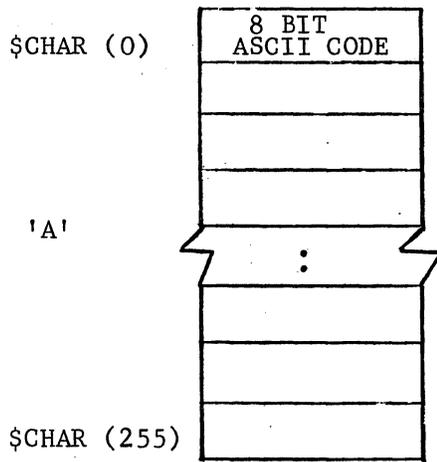


```

CONST
  LIMIT = 111111111111(2);
TYPE
  MEM = ARRAY [0..LIMIT] OF 0..LIMIT;
VAR
  MEMORY:MEM;
  :
MEMORY [0111(2)] := 010110101101(2);
MEMORY [57(8)] := 0476(8);
MEMORY [1079] := 3215;
  :

```

A 12-BIT MEMORY WITH 12-BIT ADDRESSING - FIGURE 3.12



```

CONST
  MAXVALUE = 255;
TYPE
  A=ARRAY [CHAR] OF 0..MAXVALUE;
VAR
  TRANSLATE : A,
  I : 0..MAXVALUE;
  :
I := TRANSLATE ['B'];
  :
I := TRANSLATE [$CHAR (26)];

```

A CHARACTER CONVERSION TABLE - FIGURE 3.13

SWL provides us with a couple of very powerful array operations. These are array assignment and array equality testing. No other array operations are allowed.

Array assignment allows the element by element assignment of one array to another if the arrays are the same type. This can eliminate many repetitive statements. Also, arrays may be compared for equality. Two arrays (of the same type) are equal if every pair of corresponding elements are equal. These operations are illustrated in figure 3.14.

```
VAR
  X,Y:ARRAY [1..10] OF REAL;
  :
X:=Y;
  :
IF X=Y
  THEN WRITE ('ARRAYS EQUAL',EOL);
  ELSE WRITE ('ARRAYS NOT EQUAL',EOL);
IFEND;
```

ARRAY OPERATIONS - FIGURE 3.14

## STRINGS

The string type provides a convenient way of manipulating strings of characters. We could, of course, create an array of characters. But arrays are accessed one element at a time. So an array of characters would always have to be accessed one character at a time.

Sometimes what we really want to do is manipulate a number of characters (a string of characters) as a unit. In addition, we would like to be able to examine sub-strings of various lengths without accessing each individual character.

A string may only consist of characters. Strings of other types (such as, Real, Boolean, etc.) are not allowed. Figure 3.15 shows how a string type declaration might appear.

```
TYPE
  ONELINE = STRING (72) OF CHAR;
VAR X: ONELINE;
```

STRING TYPE DECLARATION - FIGURE 3.15

In figure 3.15 the type identifier is ONE LINE. The reserved word STRING declares this type to be a string. The "(72) OF CHAR" declares the length of the string to be 72 characters. The variable X is then declared to be type ONE\_LINE.

## STRING REFERENCING

When the name (or identifier) of the string (X in figure 3.15) is used it refers to the entire string. So, X in figure 3.15 refers to a string of 72 characters.

Any individual character may be referenced by giving the position of the character in the string in parenthesis. For example X(1) is a reference to the first character of the string. X(72) would be a reference to the last character in string X.

So far, string referencing is identical to one dimensional array referencing except that the parenthesis ( ) are used in string references and the square brackets [ ] are used in array references.

But string references also include the ability to reference a sub-string of any valid length. This method of reference has no equivalent with respect to arrays. To refer to a substring the following form is used X(starting position, length). So X(1,10) would refer to the substring starting in position one and having a length of 10 characters. The length position of the reference may contain an asterisk (\*) indicating that the length should extend to the end of the string. For example X(70,\*) is a substring of the last three characters of X. We might note that there is some overlap of references. For instance, X, X(1,72) and X(1,\*) all refer to the entire string X. Similarly, X(70,3) and X(70,\*) both refer to the last three characters of the string X.

Let's put this information to use by writing a short program which reads one line from the input file and builds a string. Then the program will write the string. Remember, a string can be written, but, it can not be read. We can, however, read characters. So, our approach is to read one character at a time from the input file and build the string (one character at a time). When the string is built, it is written to the output file.

```
MODULE STRING_BUILD;
PROC [XDCL] MAIN;
CONST STRLEN = 80;
VAR LINE: STRING (STRLEN) OF CHAR,
    LEN: INTEGER,
    C: CHAR;

LEN:=0;
READ(C)
WHILE (C /=EOL) AND (NOT #EOF (INPUT)) DO
LEN:=LEN+1;
EXIT WHEN LEN >= STRLEN;
LINE (LEN):=C;
READ(C);
WHILEND;

WRITE (LINE(1,LEN),EOL);

PROCEND MAIN;
MODEND STRING_BUILD;
```

BUILDING STRINGS - FIGURE 3.16

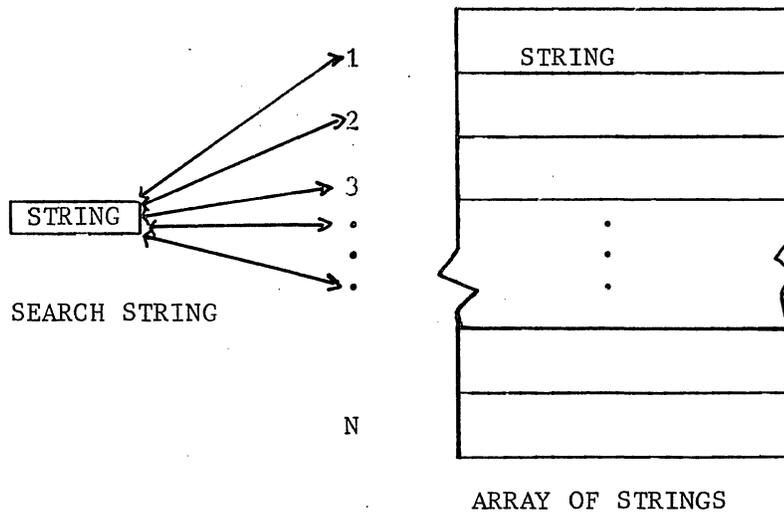
*say data has 2 lines  
len = 81  
exit*

*error  
↓  
write (LINE(1,81), EOL)*

In the program in (figure 3.16), we assume that no line (string) will exceed 80 characters. If a line does exceed 80 characters only the first 80 characters will be added to the string. Since our string length has been declared to be 80 characters, and a line may contain fewer characters we need to keep some indication (LEN) of the current (or actual) string length. The WRITE statement then writes the substring "LINE(1,LEN)" containing the actual characters read from the input file.

Another example of the use of strings is contained in the next example (figure 3.17). Here we have an array of strings. Each array element is a string of characters. The program compares a given string with all the strings in the array to see if the string can be found. A string constant may be constructed by placing a number of characters in quotes. For example, if S is a string then "S(1,3):='ABC';" assigns the string 'ABC' to the substring S(1,3).

Before looking at the program skeleton, see figure 3.17 which illustrates the problem conceptually.



STRING SEARCHING - FIGURE 3.17

Figure 3.17 illustrates the array of strings. Each string might be the name of a person, or the name of some variable, etc. The search string will be compared with each element (string) in the array.

The program skeleton to accomplish this string search is shown in figure 3.18 below.

```

MODULE  STRING_SEARCH;

PROC [XDCL] MAIN;

CONST
  SLEN = 10,
  ASIZE = 100;

TYPE
  STR = STRING(SLEN) OF CHAR;

VAR
  SARRAY : ARRAY [1..ASIZE] OF STR,
  SSTR   : STR,
  I      : 1..ASIZE+1;

"ASSUME SARRAY IS          "
"INITIALIZED BY THE PROGRAM HERE!"

"BEGIN STRING SEARCH"
SSTR := 'SMITH,JOHN';
I:=0;
REPEAT
I:=I+1;
UNTIL (SSTR=SARRAY [I]) OR (I>ASIZE)

IF I<=ASIZE
  THEN WRITE (SSTR, 'FOUND AT POSITION',I,EOL);
  ELSE WRITE (SSTR, 'NOT FOUND;EOL);
IFEND;

PROCEND MAIN;
MODEND STRING_SEARCH;

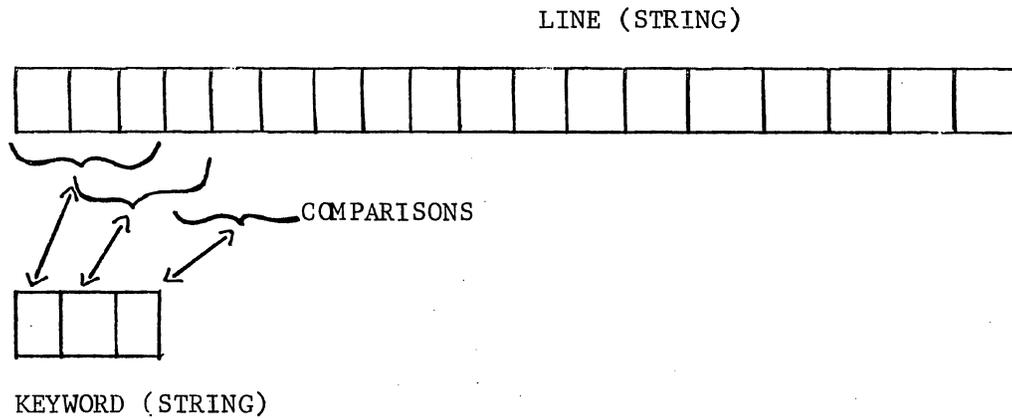
```

*10 characters*

STRING SEARCH - FIGURE 3.18

Another example of string searching uses the concept of sub-strings. This problem is to find certain keywords in a line. In SWL terminology, we want to find the occurrence of a substring in a string.

If we can assume that we have a line (string) of characters and a keyword (string), then the problem is to check each and every substring in the line for a match against the keyword string. The search is shown conceptually in figure 3.19.



FINDING ONE STRING IN ANOTHER - FIGURE 3.19

The SWL program to accomplish this search is shown in figure 3.20.

```

MODULE SUB_STRING_SEARCH;
PROC [XDCL] MAIN;
CONST
  LINELEN = 80,
  STRLEN = 3;

VAR LINE: STRING (LINELEN) OF CHAR,
    KEYW: STRING (STRLEN) OF CHAR,
    I : 1..LINELEN-2;

"ASSUME LINE IS INITIALIZED HERE"

KEYW := 'XOR';

"BEGIN SEARCH"
FOR I:= 1 TO LINELEN-2 DO
  IF KEYW = LINE (I,3)
    THEN WRITE(KEYW, 'FOUND AT POSITION", I, EOL);
  IFEND;
FOREND;
WRITE ('END OF JOB', EOL);

PROCEND MAIN;
MODEND SUB_STRING_SEARCH;

```

SUB-STRING SEARCH - FIGURE 3.20

## STRING CONVERSION <sup>1</sup>

When a string is not the right length for either an assignment or a comparison we have the task of converting the string to the correct size.

The \$STRING conversion function accomplishes the task. This conversion function can be used to lengthen (by filling on the right) or shorten (by truncating on the right) a given string.

For example, suppose we had defined a string in the following way:

```
VAR S: STRING (80) OF CHAR;
```

and we wish to initialize the string without having to write 80 blanks. We might use:

```
S := $STRING (80, ' ');
```

The \$STRING function says take the string (' ') and extend it to 80 characters. Blank fill is used by default. We can specify fill characters as shown below

```
S := $STRING (80, 'ABCD', '*');
```

## STRING REPRESENTATION <sup>2</sup>

Another thorny problem is that of converting a value into the character representation of the value. For example, converting the integer 125 into the character string '125'. This task is accomplished with the \$STRINGREP Procedure.

The function is defined as follows:

```
$STRINGREP(VALUE, SUBSTRING, WIDTH, DECIMALS)
```

Where value is the integer, real, or Boolean.

SUBSTRING is the character representation constructed by the Function.

WIDTH is the number of character positions.

DECIMALS is the number of decimal positions for real numbers.

Boolean values are converted to the string 'TRUE' or 'FALSE' RIGHT-JUSTIFIED, blank-filled.

Reals are represented as decimal numbers with exponent base 10 with "DECIMALS" digits after the decimal point.

Integer values are converted to "WIDTH" decimal digits with leading zero replaced by blanks.

1 - ISWL facility only. Automatic adjustment of string length in SWL will be accomplished using varying strings.

2 - This ISWL facility will be addressed by general formatted I/O in SWL.  
REV. A

An example is shown below:

```
VAR S: STRING (10) OF CHAR,  
    I: INTEGER;  
I:=-1276;  
#STRINGREP (I,S,10);
```

Notice that #STRINGREP is a procedure.

*set eg. in real number*

## POINTERS

SWL has a very powerful pointer capability which is used throughout the language in many ways. In this section, we will present the more elementary and straight forward uses of pointers. The other interesting uses of pointers will be covered under the topics of storage management and records.

In SWL, when a pointer is declared it is done with reference to a type. At execution time, a pointer variable can be set to point to some specific occurrence of the type. But at declaration the pointer is said to point to a type - not a variable.

The pointer symbol  $\wedge$  ( $\nearrow$  on some terminals) is used to make reference to or declare a pointer.

Some declarations of pointer types and pointer variables are shown in figure 3.21.

```
TYPE  
A = STRING (10) OF CHAR,  
B = ARRAY [1..10] OF INTEGER,  
C =  $\wedge$ A;  
  
VAR  
APTR :  $\wedge$  A,  
APTR1: C,  
BPTR :  $\wedge$  B,  
INTP :  $\wedge$  INTEGER,  
VARA : A,  
VARB : B;
```

POINTER TYPES - FIGURE 3.21

In figure 3.21, we see a number of different pointer declarations. First, C is declared to be a type which is a pointer to the type A (a string of 10 characters). Of course C is not a variable, just the name of a type. Looking at the variable declarations we see a variable APTR which is an automatic variable that is a pointer to the type A. The variable APTR1 is

also a pointer to the type A. This is so because APTR1 is type C and C is type pointer to type A. BPTR is a variable that is a pointer to type B (an array of integers). INTP is a variable that is a pointer to any integer. VARA is a variable that is type A (a string of characters).

At execution time, VARA will become a string of characters. Variable APTR (which is a pointer to a string of characters) could be made to point to VARA at execution time.

Since pointers may only be declared as pointers to some type the following

```
VAR X: INTEGER,
    Y: ^X;
```

is an error. Notice that Y is declared to be a pointer to a variable. This is a no-no. Perhaps what is needed is a pointer to X at execution time. But at declaration time, the sequence of statements would look like:

```
VAR X: INTEGER,
    Y: ^INTEGER;
```

Here Y is a pointer to type integer. X is a variable whose type is integer. At execution time, Y could be made to point to X.

#### POINTER REFERENCES

Once pointers have been declared it becomes necessary to be able to reference not only the pointer but what the pointer, points to. These problems of reference are also handled with the pointer symbol.

Another problem, is what to do with an empty pointer, i.e., the pointer doesn't point to anything.

In SWL, the reserved word NIL is used to indicate an empty pointer. Any pointer can be set to NIL. When a pointer is tested and found to be NIL, then we know that it does not point to anything.

There are four possible ways in which the ^ symbol may be used to indicate pointer references. Two of these use the ^ symbol on the right of an assignment and two use the ^ on the left of an assignment. Consider the following:

```
TYPE
  A == ARRAY [1..10] OF REAL;

VAR
  P: ^A,
  X, Y: A;

:
P := ^X;  "P IS ASSIGNED A POINTER TO X"
X := P^; "X IS ASSIGNED WHAT THE POINTER P POINTS TO"
P^ := Y;  "WHAT THE POINTER P POINTS TO IS ASSIGNED Y"
^P := X;  "THIS FORM IS ILLEGAL"
```

POINTER REFERENCES - FIGURE 3.22

Figure 3.22 illustrates the four possible uses of the  $\wedge$  symbol in assignment. Basically, the  $\wedge$  symbol to the left of a variable ( $\wedge X$ ) means "construct a pointer to X". The  $\wedge$  symbol on the right of a (pointer) variable means "use what the pointer variable points to". So, the example " $P := \wedge X$ ;" indicates that a pointer to X is found (constructed) and this pointer is assigned to P. Naturally, P must be a type of pointer to the same type that X is.

Similarly, " $P \wedge := Y$ ;" expresses that what the pointer P points to is assigned the value Y.

The form " $\wedge P := X$ " is illegal because  $\wedge P$  cannot be used on the left of an assignment.

*There is no pointer to a pointer*

In summary then:

SYMBOL	MEANING
$\wedge X$	CONSTRUCT A POINTER TO X
$P \wedge$	REFER TO WHAT P POINTS TO

POINTER SYMBOLS - FIGURE 3.23

As an example in using pointers, let's construct an array of integers and create a pointer to the array which can be used to search the array for some value (e.g. the value 3). If that value is found, we will change the value 3 to -9.

```

TYPE
  ARY = ARRAY [1..100] OF INTEGER;

VAR
  XARRAY : ARY,
  PTR    :  $\wedge$ ARY,
  I      : INTEGER;

"ASSUME XARRAY IS INITIALIZED HERE"

PTR :=  $\wedge$ XARRAY;
FOR I := 1 TO 100 DO
  IF PTR  $\wedge$ [I] = 3
    THEN PTR  $\wedge$ [I] := -9
  IFEND;
FOREND;
```

USE OF POINTERS - FIGURE 3.24

In figure 3.24 the pointer variable PTR is declared to be a pointer to type ARY. Note that variable XARRAY is type ARY. The statement "PTR:=^XARRAY;" constructs a pointer to XARRAY and assigns this pointer to the pointer variable PTR. In the IF statement, we want to check the contents of each element (integer) in the array, XARRAY. PTR is a pointer to XARRAY. PTR^ means what the pointer points to; the pointer points to XARRAY. PTR^ then is XARRAY. PTR^[3] references the third element in the array pointed to by PTR.

This interesting reference notation can be carried even further. If a pointer XPTR points to an array and the elements of the array are strings (length 10) of characters, then to reference the 2nd, 3rd, and 4th characters in the string which is the 3rd element in the array, the reference would be XPTR^[3](2,3).

↑ 2nd character  
 ↑ 3rd element in array  
RECORDS

So far all the data structures that we have discussed have been homogeneous. That is, all the elements have been identical. How does SWL enable us to create a data structure that contains non-homogeneous elements? It is done with a data structure called the record.

The record is introduced with the reserved word RECORD and terminated with RECEND. Each of the constituent elements in the record are called fields.

Figure 3.25 shows the conceptualization of a record:

FIELDS	RECORD
SURNAME	STRING (10) OF CHAR
AGE	0..100
MARRIED	BOOLEAN
SEX	BOOLEAN
FINGERS	0..11

RECEND;

THE CONCEPTUAL RECORD - FIGURE 3.25

As illustrated in figure 3.25 the record is bounded by RECORD-RECEND. Each field in the record is given an identifier (i.e. SURNAME, AGE, MARRIED, ETC.). As you can see the record can contain a mixture of types. In fact the record can contain as a field any of the types discussed so far, including another record.

If we were to commit the conceptual record of figure 3.25 to SWL syntax the result would be as shown in figure 3.26.

```
TYPE
  NAMEREC = RECORD
    SURNAME : STRING(10) OF CHAR,
    AGE : 0..100,
    MARRIED, SEX : BOOLEAN,
    FINGERS : 0..11,
    RECEND;

VAR
  RECVAR : NAMEREC,
  RECARRY: ARRAY [1..10] OF NAMEREC,
  PTR : ^NAMEREC;
```

#### RECORD SYNTAX - FIGURE 3.26

Figure 3.26 shows how the record syntax appears. In this example, the record was created as a type (NAMEREC). Subsequently, a variable RECVAR is declared that is type NAMEREC. The variable RECVAR (when assigned to storage) will contain enough space for each of the fields in type NAMEREC. The variable RECARRY is interesting. RECARRY is a ten element array. Each element in the array is a record of type NAMEREC.

#### RECORD REFERENCES

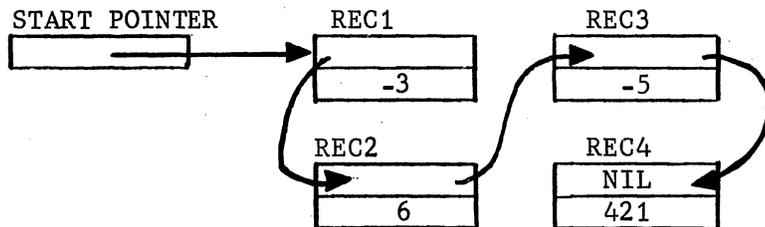
Once a variable is declared to be type record (or array of record) how do we access the individual fields of a record? Let's use the record variable RECVAR from figure 3.26. RECVAR is the variable name of a record. Whenever we use RECVAR by itself we are referring to the entire record. RECVAR.SURNAME is the way we reference the SURNAME field of the variable RECVAR. Notice the use of the period here as a field separator. The period separates the field identifier (SURNAME) from the record name (RECVAR). If we wanted to assign a surname to the record the proper assignment statement would be "RECVAR.SURNAME:='KARENSKIDY';". If we wanted to reference the first three characters of the surname in the record we might use "RECVAR.SURNAME (1,3)".

Consider the pointer PTR in figure 3.26. How would we initialize this pointer variable to point to RECVAR? Answer: "PTR:=^RECVAR;".

Once we have such a pointer, how could we use the pointer (instead of the record variable RECVAR) to initialize the age field to 15? Answer: PTR^.AGE:=15;. In this example, PTR^ points to the record and is a correct substitution for the variable name RECVAR.

Continuing, in figure 3.26 an array of records is declared as RECARRY. This array has 10 elements each one containing a record. How would we refer to the last 5 characters of the surname field of the 7th record (element) in the array? Answer: RECARRY [7]. SURNAME (6,\*);.

Our final example is a common one for programmers who deal with linked lists. How can we construct a forward pointing linked list: This problem is illustrated in figure 3.27.



A FORWARD LINKED LIST - FIGURE 3.27

Notice (in figure 3.27) how each record contains a pointer to the next record. The last record contains an empty pointer (NIL). A start pointer is needed to locate the first record in the list. Each record contains some value (shown as integers in this example).

How would we represent this interesting data structure in SWL?

```

MODULE LINKED_LIST;
PROC [XDCL] MAIN;
TYPE
  REC = RECORD
    FPTR: ^REC,
    VALU: INTEGER,
    RECEND;
VAR
  STARTPTR: ^REC,
  REC1, REC2, REC3, REC4 : REC;
  :
  STARTPTR ::= ^REC1;
  REC1. FPTR := ^REC2;
  REC2. FPTR := ^REC3;
  REC3. FPTR := ^REC4;
  REC4. FPTR := NIL;
  REC1. VALU := -3;
  REC2. VALU := 6;
  REC3. VALU := -5;
  REC4. VALU := 421;
  :
PROCEND MAIN;
MODEND LINKED_LIST;
  
```

LINKED LIST IN SWL - FIGURE 3.28

The example in figure 3.28 is a nice illustration, but lacking in its applicability to real programming practice. From this example we can see how to construct the structure shown in figure 3.27. The problem with figure 3.28 is that our program states explicitly how many records exist (in this example - four). In a real programming problem the number of records would change dynamically and we would have to add new records onto our list and delete records from the list, as required by the dynamic execution of the program.

This dynamic allocation and freeing of records (or space) will be covered later under the topic of storage management.

## SETS

The SWL notion of a set follows reasonably closely the mathematical concept of set. A set is simply an accumulation of elements (members). The elements can be identifiers (an ordinal), characters, boolean, or integers. The set members have no order. We cannot say that some member is the first member. However, we can place members in the set, delete members from the set, and determine whether a given member exists at the present time in the set. Of course, we can create both empty and full sets.

Figure 3.29 illustrates the kinds of sets allowed and the syntax for declaring sets.

```
TYPE
  A = SET OF (RED, GREEN, BLUE), "ORDINAL SET"
  B = SET OF 0..17, "INTEGER SET"
  C = SET OF 'A'..'Z'; "CHARACTER SET"

VAR
  ORDSET : A,
  INTSET : B,
  CHRSET : C;
```

### DECLARATION OF SETS - FIGURE 3.29

In figure 3.29, type A is a set of the ordinal identifiers (RED, GREEN, BLUE). That means that there are three possible members for the set type A. When a variable, such as, ORDSET is declared, it is a set variable. If ORDSET is empty then it contains no members. If ORDSET is full then the members RED, GREEN, and BLUE are all present in the set.

The obvious question then is how do we create empty and full sets? How do we place members in the set and remove members from the set?

We use a conversion function to convert identifiers (or constants) into set members. For example (using figure 3.29) \$A[RED] is a conversion

function that converts the identifier RED into the set element RED belonging to the set type A. Since we have a variable ORDSET, we could initialize (or assign) the member RED to the variable ORDSET by writing: "ORDSET := \$A[RED];". If we wanted to initialize (or assign) two members into ORDSET we could write: "ORDSET := \$A[RED,BLUE];".

The empty set is constructed by having no members between the square brackets. For example (using figure 3.29) to make the variable INTSET empty we could write: INTSET := \$B[];. If we wanted to place all the members in CHRSET we could use one of two methods

- 1) create an empty set and complement the empty set (giving a full set).

For example:

```
CHRSET := NOT $C[];  
OR  
CHRSET := NOT (CHRSET XOR CHRSET);  
OR  
CHRSET := NOT CHRSET XOR CHRSET;
```

- 2) write out each and every member of the set.

For example:

```
CHRSET := $C['A', 'B', 'C', 'D', 'E', 'F', 'G',  
            'H', 'I', 'J', 'K', 'L', 'M', 'N',  
            'O', 'P', 'Q', 'R', 'S', 'T', 'U',  
            'V', 'W', 'X', 'Y', 'Z'];
```

#### MAKING FULL SET - FIGURE 3.30

After seeing the two methods you would probably choose the first method whenever number of members in the set is large.

Now that we know how to assign members to sets we should look at the kind of set operations that exist in SWL. These operations are summarized by example in figure 3.31 below.

```

TYPE
  A = SET OF 0..5;
VAR
  SETA, SETB, SETC:A;
  B: BOOLEAN;
  .
  .
  "RESULT"
"ASSIGN MEMBERS TO A SET"
SETB := $A[2,0,1];
"0,1,2"

"INCLUDE OR OF TWO SETS"
SETC := SETB OR $A[4,0,1];
"0,1,2,4"

"LOGICAL PRODUCT (AND) OF SETS"
SETA := SETB AND SETC;
"0,1,2"

"SET IDENTITY (or equality)"
B := SETA = SETC;
"FALSE"

"SET MEMBERSHIP"
B := 5 IN SETA;
"FALSE"

"SET INEQUALITY"
B := SETA /= SETC;
"TRUE"

"SETA IS CONTAINED IN SETC"
B := SETA <= SETC;
"TRUE"

"SETA CONTAINS SETC"
B := SETA >= SETC;
"FALSE"

"SET DIFFERENCE, THE SET"
"CONSISTING OF ELEMENTS OF THE"
"LEFT OPERAND THAT ARE NOT ALSO"
"ELEMENTS OF THE RIGHT OPERAND"
SETA := SETC - $A[1,5];
"0,2,4"

"SYMMETRIC DIFFERENCE, THE SET"
"OF ELEMENTS CONTAINED IN EITHER"
"SET BUT NOT BOTH SETS"
SETA := SETC XOR $A[1,5];
"0,2,4,5"

```

SET OPERATION EXAMPLES - FIGURE 3.31

✓ STORAGE MANAGEMENT

We are often faced with the programming problem of managing storage efficiently. We want to be able to use the storage space efficiently and to access the space efficiently. The statements presented in this section help accomplish these goals.

THE UNIVERSAL HEAP

The universal heap is an area associated with the users program storage that can be managed explicitly by the user. Two statements are provided. The ALLOCATE statement is used to assign (or use) space in the universal heap. When the user allocates some type to the universal heap, SWL reserves enough space for the type and returns a pointer to the space to the user. Using the pointer, then, the user may access the space, store information into the space and retrieve data previously stored into the space. If the space cannot be allocated the pointer is set to NIL.

When the user no longer needs the reserved (allocated) space the FREE statement may be used to make the space available for use later on in the program execution.

Any data type may be allocated explicitly by the user. In practice, however, the array, record, and sequence (to be discussed shortly) types are most often allocated.

Lets examine the use of explicit allocation with respect to RECORDS. In this example, we will create a linked list by allocating space in the universal heap. See figure 3.27 for a diagram of the forward linked list.

```

PROC [XDCL] MAIN
TYPE
  R = RECORD
    FPTR:AR,
    VALU: INTEGER;
VAR
  P, TEMPTR:AR;
  N: INTEGER;

  READ(N);                                "FIRST ALLOCATE"
  ALLOCATE P;
  IF P=NIL
    THEN WRITE ('NO ALLOCATE',EOL);
    ELSE P^.FPTR:=NIL;
        P^.VALU:= N;
  IFEND;
  :
  :
  READ (N);                                "SECOND ALLOCATE"
  ALLOCATE TEMPTR;
  IF TEMPTR = NIL
  THEN WRITE ('NO ALLOCATE',EOL);

```

```

ELSE  P^.FPTR:=TEMPTR;
      TEMPTR^.FPTR:=NIL;
      TEMPTR^.VALU:=N;
IFEND

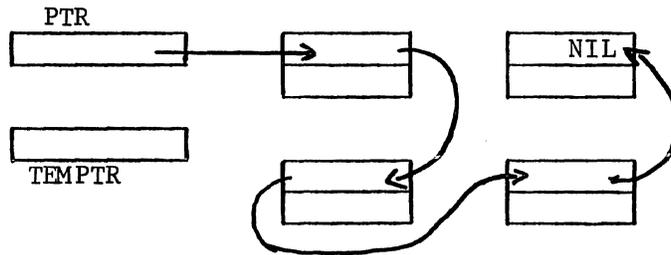
```

STORAGE ALLOCATION - FIGURE 3.32

Naturally, these allocate statements would be in some repetitive statement - not written in line, as shown in the example in figure 3.32.

If we were through using our linked list and wanted to free all the allocated records we would have to work our way down the linked list freeing each record in turn until we reached the record containing a forward pointer of NIL.

This successive freeing is illustrated in figure 3.33.

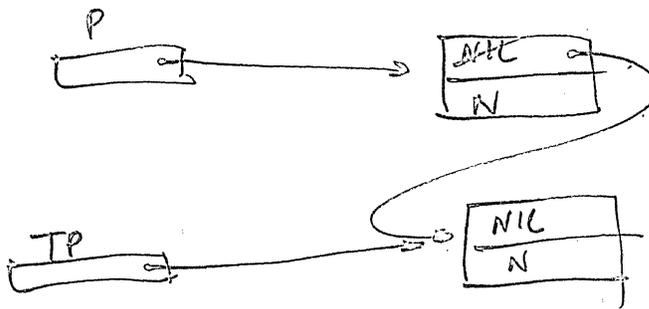


```

:
TEMPTR :=PTR
WHILE TEMPTR /=NIL DO
PTR := TEMPTR^.FPTR;
FREE TEMPTR;
TEMPTR :=PTR;
WHILEND;
:

```

FREEING A LINKED LIST - FIGURE 3.33



Allocate Sequences (Fig 3.32)

## SEQUENCES

The sequence type represents a storage structure whose components are referenced through sequential accessing. Once a sequence (variable) is declared there are only two operations that can be performed, NEXT and RESET. This sequential access to storage is analogous to the sequential file access provided by the READ statement.

When a sequence is declared a definite measure of the space required by the sequence must be provided. The NEXT statement is used both to enter values into the sequence and to remove (or obtain) values from the sequence. The reset statement may be used to place the sequence pointer so that the NEXT statement will access the first element in the sequence.

When declaring the space required by a sequence the form "REP N OF type" is used. That is the sequence must be large enough to hold N repetitions of type. The example in figure 3.34 illustrates how we use the sequence as a data type.

```
MODULE SEQUENCE_TEST;
PROC [XDCL] MAIN;
TYPE
  A = SEQ(REP 10 OF INTEGER);
VAR
  P: ^INTEGER,
  ASEQ : A,
  I : INTEGER;

  "PLACE 10 VALUES IN ASEQ"
  FOR I := 1 TO 10 DO
  NEXT P IN ASEQ;
  P^:=I*5;
  FOREND;

  "WRITE ALL VALUES IN SEQUENCE"
  RESET ASEQ;
  NEXT P IN ASEQ;
  WHILE P/= NIL DO
  WRITE (P^:5,EOL);
  NEXT P IN ASEQ;
  WHILEND;

  PROCEND MAIN;
MODEND SEQUENCE_TEST;
```

SEQUENCE SYNTAX - FIGURE 3.34

The statement "NEXT P IN ASEQ;" provides a pointer to the next element in the sequence ASEQ. To access this element we must use the pointer reference PA. So, to place a value in the sequence we use a statement like "PA:=1\*5;"

To access the values in the sequence first RESET the sequence and then obtain successive pointers to elements in the sequence with the "NEXT P IN ASEQ;" statement executed repetitively. Notice that when the end of the sequence is reached, the "NEXT P IN ASEQ" will return a pointer value that is NIL (indicating the end of the sequence).

### VARIABLE ATTRIBUTES

When variables are declared certain attributes may be associated with the variable. We will consider two classes of attributes: 1) storage attributes and 2) scope attributes.

Recall that the general form of a variable declaration is "VAR identifier: type;". When we wish to add attributes to the variable declaration the form becomes "VAR identifier: [attributes] type;". That is, the attributes are added after the colon (:) and before the type and the attributes are enclosed in square brackets.

### STORAGE ATTRIBUTE

A variable declared without any attributes becomes an automatic variable. Storage for this type of variable is allocated automatically when the block in which the declaration exists is entered (at execution time). The storage for such a variable is freed upon EXIT from the block in which it is declared. When a variable is given the [STATIC] attribute the automatic allocation is not changed. But, the storage for the variable is not freed upon exit from the block. In fact, once allocated the storage for the variable remains allocated for the lifetime of the program. This can be very helpful for counters (for instance) where the counter should not be re-initialized (or re-allocated).

### SCOPE ATTRIBUTES

The scope attributes are [XDCL] and [XREF]. Whenever either of these attributes are used the variable is by default [STATIC].

The [XDCL] attribute indicates that the variable is declared, in this module, and this declaration is available to other modules. *Declared External Available*

The [XREF] attribute indicates that the variable is used (referenced) in this module and that the actual variable declaration is contained in some other module.

Prior to program execution the [XREF] and [XDCL] variables (with the same identifier) are bound together so that only one occurrence of the variable actually exists in storage. To effect this "linkage" the [XREF] and [XDCL] variable declarations should express the same type.

The example in figure 3.35 illustrates the use of [XREF] and [XDCL] in two modules.

```

MODULE PASS1;
:
VAR SYNTAX TABLE
  : [XDCL] ARRAY
  [1..100] OF CHAR;

"VARIABLE SYNTAX TABLE"
"DECLARED XDCL HERE SO THAT"
"IT CAN BE ACCESSED   "
"FROM ANOTHER MODULE  "
:
MODEND PASS1;

MODULE PASS2;
:
VAR SYNTAX TABLE
  : [XREF] ARRAY
  [1..100] OF CHAR;

"VARIABLE SYNTAX TABLE"
"DECLARED XREF SO THAT "
"THE ACTUAL STORAGE    "
"ALLOCATED IN PASS1 WILL BE USED"
:
MODEND PASS2;

```

XDCL and XREF ATTRIBUTES - FIGURE 3.35

As shown in figure 3.35 only one, one-hundred-element array will be allocated storage. This storage allocation is from MODULE PASS1. Any use of the variable SYNTAX TABLE from MODULE PASS 2 will really be referencing the storage allocated in MODULE PASS1.

If variables are declared and used in only one module, then there will be no need for the [XDCL] and [XREF] attributes.

Our example shows these attributes with an array variable. Of course, any type of variable can be made [XDCL] or used as [XREF]

### PACKED VARIABLES

The ability to specify that a variable is packed, is the SWL method for providing the programmer a choice between compacting data and access time. Generally, a packed variable (such as a packed array) will save space but require greater time to access any individual element.

There are only two data types that can be packed - arrays and records.

Although the actual packing algorithm may differ from one language implementation to another the concept of the trade-off between storage space and access time is a valid one

Take a look at the example in figure 3.36.

```
VAR
  X : ARRAY [1..100] OF 0..15,
  Y : [PACKED] ARRAY [1..100] OF 0..15;
```

#### PACKED DATA - FIGURE 3.36

In figure 3.36, both arrays contain one hundred elements. In both cases the array contents is in the range 0 to 15. The variable X will optimize access time at the expense of storage space. The variable Y will optimize storage space at the expense of longer access time. It's your choice - access time or efficient storage. Generally, you can't have both.

The actual packing algorithm (or number of bits used) need not concern us here.

#### VARIABLE INITIALIZATION

Occasionally, it would be nice if we could initialize a variable along with the declaration of the variable. That is, not have to write a separate statement to perform the initialization. At the present time only [STATIC] variables may be initialized.

To initialize a variable we simply add an assignment onto the end of the variable declaration.

Some simple instances of variable initialization are shown in figure 3.37.

```
VAR
  X : [STATIC] INTEGER := 0,
  Y : [STATIC] BOOLEAN := TRUE,
  R : [STATIC] REAL := 1.537,
  Z : [STATIC] (RED, BLUE) := RED,
  C : [STATIC] CHAR := 'T';
```

#### VARIABLE INITIALIZATION - FIGURE 3.37

Of course more complicated data types may be initialized too. For instance, RECORDS, ARRAYS, and SETS may be initialized. When arrays are initialized the values for each dimension are enclosed in square brackets []. When records are initialized each record is enclosed in square brackets.

As an example, let's initialize a two-dimensional array as shown in figure 3.38.

# Initialization =

```
VAR
  X : [STATIC] ARRAY [1..2] OF
        ARRAY [3..5] OF INTEGER
      := [[6,7,-3], [5,7,29]];
```

## ARRAY INITIALIZATION - FIGURE 3.38

Notice that array X is an array of an array. Each element in the first array is an array of three elements. So, two groups of three integers are required for initialization. The initialization expression `[[ ] [ ] ]` contains two groups of three integers.

As one last example of initialization consider the variable R which is a record containing a record, as shown in figure 3.39.

```
VAR
  R : [STATIC] RECORD
        AGE: 6..66,
        MARRIED, SEX: BOOLEAN,
        DATE: RECORD
              DAY: 1..31,
              MONTH: 1..12,
              YEAR: 70..80
            RECEND
        RECEND
      := [23, TRUE, TRUE, [3, 5, 73]];
```

## RECORD INITIALIZATION - FIGURE 3.39

In figure 3.39, each record initialization is enclosed in square brackets `[ ]`. The initialization values are explained in figure 3.40.

<u>FIELD</u>	<u>VALUE</u>
R. AGE	23
R. MARRIED	TRUE
R. SEX	TRUE
R. DATE.DAY	3
R. DATE.MONTH	5
R. DATE.YEAR	73

## RECORD INITIALIZATION VALUES - FIGURE 3.40

Sets may be initialized, too, by placing the initialized set members in square brackets [] as shown below in figure 3.40.1

```
VAR  
S : [STATIC] SET OF 0..9  
  := [8,4,9,7];
```

#### SET INITIALIZATION - FIGURE 3.40.1

Strings may be initialized by placing the initialization values in single quotes. Also, the CAT operator may be used in string initialization as shown below:

```
VAR  
S1 : [STATIC] STRING (5) OF CHAR := 'ABCDE',  
S2 : [STATIC] STRING (10) OF CHAR  
  := 'STUVW' CAT 'AB5ØC';  
STRING INITIALIZATION - FIGURE 3.40.2
```

NOTE Pointers may not be initialized.

#### TYPE CONVERSION

The operations of assignment, comparison, and arithmetic are defined only for operands of equivalent types. When it is necessary to operate on operands that do not meet the requirements of type equivalence the type conversion functions must be used.

There are two classes of type conversion. The primitive conversions are described on page 2-15. Structured conversions are not available in ISWL but will be provided in SWL.

## I/O REVISITED

In our earlier discussions of Input and Output (the READ and WRITE statements) we did not discuss the problem of reading more than one line of data from a terminal. Our previous I/O discussions were correct for reading data from a data file. When we have a data file, all the data exists in the file at the time the file is opened (if it is a file to be read).

In contrast to this, terminal input is entered one line at a time. Each line entered is in effect a one line file. So to read a second line from a terminal we must rewind the terminal input file with the statement "REWIND(INPUT);".

As an overview, then, we can see that a program written to read data from a file will not in general read terminal input successfully. And, a program written to read terminal data will not work correctly when the data is provided via an existing file. The difference between the two programs will be the "REWIND(INPUT);" statement.

Assume that an ISWL program intends to read four (4) lines of data. The data is integer and we wish to sum up the integer values.

An ISWL program to accomplish this from a data file would have the following statements:

```
MODULE FILEIO;
PROC [XDCL] MAIN;
VAR I,S:INTEGER
S:=0;
READ(I);
WHILE NOT #EOF(INPUT)DO
S:=S+I;
READ(I);
WHILEND;
WRITE(S,EOL);
PROCEND MAIN;
MODEND FILEIO;
```

### READING INPUT FROM DATA FILE - FIGURE 3.41

In figure 3.41 the program will read data until it reaches the end of file. All read statements look like they are reading from file input however, using control language statements we will (at load time) change file input to some data file.

To read a similar four lines from a terminal the following program could be used:

```
MODULE TERMINALIO;
PROC [XDCL] MAIN;
VAR I,S : INTEGER;
S:=0;
REPEAT
READ(I);
WHILE NOT #EOF(INPUT) DO
S:=S+I;
READ(I);
WHILEND;
REWIND(INPUT);
UNTIL #EOF(INPUT);
WRITE(S,EOL);
PROCEND MAIN;
MODEND TERMINALIO
```

#### READING INPUT FROM TERMINAL - FIGURE 3.42

In figure 3.42 the program reads data until it reaches end-of-file. This condition really signals end-of-line since each line is terminated by an end-of-file. Then the program does "REWIND(INPUT);" which gets set for the next line. If the terminal user keys in an empty line (carriage return only) the end-of-file will be set and the program will exit the REPEAT statement.

Actually, the program of figure 3.42 could be used for file I/O if the "REWIND(INPUT);" statement was removed.



## CHAPTER 4

### ADVANCED SWL

This chapter includes discussions about the more advanced features supported by SWL. The material presented will cover Procedures, Functions, Adaptable types, Unions, Conformity case, Variant records, Labels, Control Statements, Advanced I/O, Representation Dependent facilities, Standard Functions and Compile Time options.

## PROCEDURES

Before studying this section you may wish to review the material on Procedures, Block Structure, Call by Value, Call by Reference, and Scope of Identifiers presented in Chapter 1.

A procedure is a convenient way of declaring some variables and execution time statements in one place in a module and referring to (or calling) the procedure from some other location in the module or from some other module.

A procedure is delimited by the PROC and PROCEND statements. The simplest example of a procedure declaration is a procedure that has no parameters passed to it. This form of procedure normally uses global variables to obtain or store data.

This very simple procedure declaration is shown in figure 4.1.

```
      :  
      "GLOBAL VARIABLE DECLARATION"  
      VAR I,J,K:INTEGER;  
      :  
      "PROCEDURE DECLARATION"  
      PROC ADD;  
          K:=I+J;  
      PROCEND ADD;  
  
      "PROCEDURE CALL"  
      PROC [XDCL] MAIN  
          READ (I,J);  
          ADD;  
          WRITE (K,EOL);  
      PROCEND MAIN;
```

SIMPLE PROCEDURE DECLARATION & CALL - FIGURE 4.1

The example in figure 4.1 illustrates both the simple procedure declaration and the procedure call. In this example, variables I, J, and K are global variables. Their scope includes both procedure ADD and procedure MAIN. Procedure ADD, when called, adds the global variable I to the global variable J and assigns the result to global variable K. Note that no parameters are actually passed to the procedure ADD directly.

In procedure MAIN, figure 4.1, the first statement reads values for the global variables I and J. Then the procedure ADD is called (by the line "ADD;"). Procedure ADD does its computation and returns (at the "PROCEND ADD;" statement) to procedure MAIN. The next statement in procedure MAIN writes the results (K).

This example illustrates the declaration and call to a procedure using the global variable mechanism for communication. This mechanism provides good communication, but does not provide much protection for the global variables.

Next, we want to describe the passing of parameters to a procedure. When a procedure is declared, and parameters are specified in the declaration, these parameters are called FORMAL parameters.

When a procedure is called and parameters are placed in the call the parameters are known as ACTUAL parameters. These concepts are shown in figure 4.2.

```
PROC ADD (FORMAL PARAMETERS);  
:  
PROCEND ADD;  
}  
PROC [XDCL] MAIN;  
:  
ADD (ACTUAL PARAMETERS);  
:  
PROCEND MAIN;
```

ACTUAL and FORMAL PARAMETERS - FIGURE 4.2

The formal parameters and actual parameters must agree in number, order, and type.

The formal parameters also declare whether parameter passing is done by value or by reference. If the parameter passing is by value then a copy of the actual parameter is made for the procedure which is called. This mechanism insures that the procedure called will have the value of the actual parameter and that the procedure called cannot change the actual parameter. This provides protection for the actual parameter.

If the formal parameter indicates a call by reference then a pointer to the actual parameter is passed to the procedure called. In this manner, the called procedure can read and alter the actual parameter.

It should be clear that a call by reference parameter provides excellent communication but provides limited protection (or security). The call by value parameter provides excellent protection (or security) but poor communication.

Now we can modify the simple add procedure of figure 4.1 to include both call by reference and call by value parameters. Notice the syntax of the parameters.

```

MODULE PROC_WITH_PARAMETERS;
PROC ADD (REF M:INTEGER;
          VAL N,P:INTEGER;);
M:=N+P;
PROCEND ADD;

PROC [XDCL] MAIN;
VAR I,J,K:INTEGER;
READ(J,K);
ADD(I,J,K);
WRITE(I,EOL);
PROCEND MAIN;
MODEND PROC_WITH_PARAMETERS;

```

#### PROCEDURE WITH PARAMETERS - FIGURE 4.3

In the example in figure 4.3 notice that in the procedure declaration for ADD three (formal) parameters are declared. The first (M) is call by reference and the last two (N,P) are call by value. The procedure then uses the variables M, N and P to compute some result.

Referring now to procedure MAIN in figure 4.3 notice the local variables I,J and K. These variables cannot be accessed by procedure ADD directly because the scope of the identifiers (I,J,and K) is limited to procedure MAIN.

At execution time the first statement executed will be the "READ(J,K);" statement. The next statement "ADD(I,J,K);" calls the procedure ADD and passes the three parameters. Since the 2nd and 3rd parameters are call by value (see PROC ADD declaration) a copy of the actual parameters (J,K) is made for the procedure ADD. Inside procedure ADD these two paramters are known as N and P. From the viewpoint of procedure ADD variables N and P are local variables and they are copies of the actual parameters J and K.

The actual parameter I, however, is different. In the procedure ADD declaration the 1st (formal) parameter is declared to be call by reference. This means that procedure add will receive a pointer (reference) to the actual parameter (I). Notice that this means that any changes (assignments) to the 1st formal parameter within PROC ADD will modify the actual parameter (in this case I).

The call by reference for the 1st formal parameter is essential in this example because it provides the mechanism for the procedure ADD to return the result to the procedure MAIN.

After procedure ADD is completed (the "PROCEND ADD;" statement is executed) control returns to the write statement in procedure MAIN and the result I is written to the output file.

What would happen if the 1st formal parameter (M) had been declared to be call by value instead of call by reference. This is illustrated below:

```
PROC ADD (VAL M:INTEGER;  
          VAL N,P:INTEGER;);  
M:=N+P;  
PROCEND ADD;
```

In this case, when the procedure add was called, a copy of the actual parameter (I, which is undefined) would be made for the procedure to use. After the procedure ADD statement "M:=N+P;" is executed, the result value M would be a local value (local variable). When procedure ADD returns to Procedure MAIN, all local variables will be destroyed. So, the value of M we so carefully computed would be destroyed. The actual parameter (I) would remain undefined and the write statement would write out some undefined quantity for the value of I.

It should now be apparent that we must carefully declare formal parameters according to their use considering both communication and protection of variables.

#### FUNCTIONS

Functions are similar to procedures. Their declaration is almost identical with the exception of a return type, added on at the end, that declares the type of the function. Functions normally return a value. This value can then be used in other computations.

Because the function returns a value, we cannot place the function call on a line by itself (what would happen to the result?). Therefore, a function call is normally embedded in some other statement (such as an assignment or IF statement).

Let's write our ADD routine as a function and see how it differs from the procedure use of the same routine.

```
MODULE FUNCTION_ADD;  
PROC ADD (VAL N,P:INTEGER) INTEGER;  
ADD:=N+P;  
PROCEND ADD;  
  
PROC [XDCL] MAIN;  
VAR I,J,K:INTEGER;  
READ (J,K);  
I:=ADD(J,K);  
WRITE(I,EOL);  
PROCEND MAIN;  
MODEND FUNCTION_ADD;
```

FUNCTION DECLARATION - FIGURE 4.4

In the example in figure 4.4 notice the declaration of the function ADD. It looks just like the procedure declaration except for the reserved word "INTEGER" at the end of the line "PROC ADD (VAL N,P:INTEGER) INTEGER;". This word does two things: first, it specifies that this is a function declaration (instead of a procedure declaration); and second, it specifies that the function returns a type which is an integer. Or saying it differently, the function returns an integer value.

Now, inside the function "ADD", the line that computes the result "ADD:=N+P;" adds N+P and assigns the result to the function name! Yes, this is how the function obtains its value.

Looking at the call to the function (from procedure MAIN) "I:= ADD(J,K);", we see that the actual parameters J and K are sent to the function and that "ADD(J,K);" must result in some value because the value is assigned to I. The value of "ADD(J,K)" is called the function value. You can see how the function name in the call is part of a larger statement.

One final comment on the restrictions on functions. Functions can not return all call types. A function can return an integer, character, ordinal, boolean, real, or pointer result.

---

## NESTED PROCS and FUNCTIONS

When procedures or/and functions are nested, we obtain security for the function or procedure itself as distinct from the parameters which are passed.

For example, consider the program structure in figure 4.5.

```
MODULE NESTED_PROCS;
  PROC COMPUTE;
    VAR
      PROC TEST;
      VAR
        "EXECUTABLE STATEMENTS HERE"
      PROCEND TEST;
      "EXECUTABLE STATEMENTS HERE"
    PROCEND COMPUTE;

  PROC [XDCL] MAIN;
  :
  COMPUTE;
  :
  PROCEND MAIN;

MODEND NESTED_PROCS;
```

] COMPUTE  
] MAIN

NESTED PROCS - FIGURE 4.5

In the example in figure 4.5 we see that procedures MAIN and COMPUTE are at the outermost level. Procedure TEST is nested inside COMPUTE. This structure provides some shielding (protection) for procedure TEST.

With this structure, procedure MAIN can call procedure COMPUTE, BUT procedure MAIN cannot call procedure TEST. Only COMPUTE can call TEST. So TEST can rely on COMPUTE to validate variables and perform other computations which might be necessary for the correct execution of TEST.

This kind of nesting structure is frequently used to provide protection for procedures or/and functions.

## XDCL and XREF ATTRIBUTES

We discussed XDCL and XREF attributes with respect to variables in Chapter 3. The concept is equivalent for procedures and functions. These attributes are only needed when we wish to declare a procedure (or function) in one module and be able to refer to (or call) that procedure (or function) from another module.

Figure 4.6 below provides an example of this kind of referencing mechanism.

```
MODULE FIRST;
/PROC [XREF] COMPUTE;

PROC [XDCL] MAIN;
:
:
COMPUTE;
:
:
PROCEND MAIN;
MODEND FIRST;

MODULE SECOND;
/PROC [XDCL] COMPUTE;
:
:
PROCEND COMPUTE;
MODEND SECOND;
```

#### XDCL and XREF PROCEDURES - FIGURE 4.6

In figure 4.6 modules FIRST and SECOND would each be separately compiled and then loaded together. Note that procedure COMPUTE is declared in module SECOND. In fact, no other declarations exist in module SECOND. The procedure COMPUTE (in module SECOND) is given the attribute XDCL. This means that the procedure is declared in this module and can be referenced from some other module.

In module FIRST only the line "PROC [XREF] COMPUTE;" appears to identify COMPUTE. This line indicates that COMPUTE is a procedure. The XREF attribute indicates that the procedure (COMPUTE) will be referenced in this module (FIRST) and is declared somewhere else. Inside module FIRST, the identifier COMPUTE is known as a procedure name.

If procedure COMPUTE had any parameters, the parameter list (formal parameters) would have to be specified in each procedure declaration in each MODULE. And, the parameter specifications would have to agree in number, order, and type. The identifiers of corresponding parameter could, however, be different.

We might also note that "PROC [XDCL] MAIN;" declares a procedure whose name is MAIN that is XDCL (declared here for reference elsewhere). This is a special procedure. The loader (a procedure) calls the procedure MAIN to begin program execution. Hence, we must have one procedure MAIN and it must be XDCL so the loader can call it.

## ADAPTABLE TYPES

Before we cover adaptable types lets examine a SWL function to sum (add) all the elements in an array and return the sum as a result.

```
MODULE SUMMATION;

TYPE Q = ARRAY [1..10] OF INTEGER;
PROC SUM (VAL VECTOR : Q) INTEGER;
  VAR I: INTEGER;
  SUM := 0;
  FOR I:= 1 TO 10 DO
    SUM := SUM + VECTOR [I];
  FOREND;
PROCEND SUM;

PROC [XDCL] MAIN;
VAR K: ARRAY [1..10] OF INTEGER,
    RES,I: INTEGER;
FOR I:= 1 TO 10 DO
  READ (K[I]);
FOREND;
RES:= SUM(K);
WRITE (RES,EOL);
PROCEND MAIN;
MODEND SUMMATION;
```

SIMPLE SUMMATION FUNCTION - FIGURE 4.7

From the structure of the program in figure 4.7 we can see that procedure MAIN calls (and uses) the function SUM. Notice that in MAIN an array (K) is declared to contain ten elements. Because MAIN calls SUM, SUM must also contain a declaration for an array (VECTOR). The annoying thing is that the array vector (type Q) also contains ten elements.

Why is this a problem? If procedure MAIN were altered to read in twenty integers (and the variable declaration for K were also changed to be 1..20) then we would have to change the function declaration also. In this example, the statement "TYPE Q = ARRAY [1..10] OF INTEGER;" would have to be changed to an array of 20 elements.

This change of the function SUM is unfortunate. It would be much nicer if the function could change the number of elements in the array to suite (or conform to) the number of elements in the actual parameter.

There is such a type in SWL and it is called the ADAPTABLE type. It is called ADAPTABLE because the formal type can adapt (to a limited extent) to the specification of the actual parameter. This process of ADAPTING takes place at execution time. We say "to a limited extent" because ADAPTABLE means adjustable size or bounds. The type of parameter cannot be adjusted.

In our example (figure 4.7) if the variable VECTOR were adaptable then we would have some problem with the statement "FOR I:= 1 TO 10 DO". If the variable VECTOR were adaptable its first element might not be one. Further, its last element would almost certainly not be ten. Our FOR statement must be able to adapt too. The problem can be expressed as one of being able to find out what the current bounds of an adaptable array are. In SWL, two functions exist to help us. The #LOWERBOUND (adaptable array name, dimension) function returns an integer which is the lowerbound of the adaptable array in the specified dimension. This dimension parameter makes it possible to use the lowerbound function to find the lowerbound in any dimension of the adaptable array. Similarly, the #UPPERBOUND (adaptable array name, dimension) function is used to determine (at execution time) the upperbound of an adaptable array.

Using these concepts we can rewrite the MODULE as shown in figure 4.8.

```

MODULE BETTER_SUMMATION;

TYPE Q =ARRAY [*] OF INTEGER;
PROC SUM (VAL VECTOR:Q) INTEGER;
VAR I: INTEGER;
SUM:= 0
FOR I:= #LOWERBOUND (VECTOR,1) TO
        # UPPERBOUND (VECTOR,1) DO
SUM:= SUM + VECTOR [I];
FOREND;
PROCEND SUM;

PROC [XDCL] MAIN;
VAR K: ARRAY[1..10] OF INTEGER;
RES,I: INTEGER;
FOR I:= 1 TO 10 DO
READ (K[I]);
FOREND;
RES := SUM (K);
WRITE (RES,EOL);
PROCEND MAIN;
MODEND BETTER_SUMMATION;

```

#### ADAPTABLE SUMMATION FUNCTION - FIGURE 4.8

In the example in figure 4.8, notice the statement "TYPE Q=ARRAY [\*] OF INTEGER;". The asterisk [\*] indicates that the array is adaptable and that the indices are integer. Inside the function SUM notice the way the FOR statement has been written to take on the lowerbound and upperbound of the adaptable array.

If we wish to read in a different number of integers, now we need only change procedure MAIN. No changes will be needed for the function SUM.

We have succeeded in writing a more general purpose function.

## ADAPTABLE SPECIFICATIONS

Adaptable types are used only as formal parameters for procedures and functions. There are six data types which can be adaptable. These are ARRAYS, STRINGS, RECORDS, STACKS, SEQUENCES, and HEAPS.

An adaptable array can be specified in a number of ways. Adaptable array indices are not limited to integers. Figure 4.9 illustrates some of the adaptable types allowed for arrays.

### TYPE

"ADAPTABLE ARRAY WITH INTEGER INDICES"

Q = ARRAY [\*] OF REAL,

"ADAPTABLE 2-D ARRAY WITH INTEGER INDICES"

R1 = ARRAY [\*] OF ARRAY [\*] OF BOOLEAN,

R2 = ARRAY [\*,\*] OF BOOLEAN,

"PARTIALLY ADAPTABLE ARRAY WITH INTEGER INDICES"

S1 = ARRAY [3..\*] OF CHAR,

S2 = ARRAY [\*..53] OF REAL,

"ADAPTABLE ARRAY WITH CHARACTER INDICES"

T = ARRAY [\*:CHAR] OF REAL;

### ADAPTABLE ARRAY SPECIFICATIONS - FIGURE 4.9

In addition to adaptable arrays we may have adaptable strings. The string declaration is simple in that the string has only one type, character, and one length specification. In addition, the length specification must be integer. Figure 4.10 illustrates the adaptable string specification.

### TYPE

STR = STRING (\*) OF CHAR;

### ADAPTABLE STRING SPECIFICATION - FIGURE 4.10

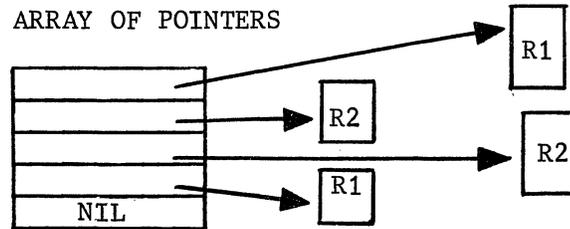
When we use adaptable strings inside a procedure or function we need a method of obtaining the string length of the actual parameter. The #STRLENGTH (adaptable string variable name) function returns an integer value which is the length of the adaptable string variable name. This function is the string equivalent of the upper and lower bound functions for arrays.

The record type may also be adaptable if one and only one of its fields are of adaptable type.

## UNION OF TYPE

Sometimes we have a need for a variable that can contain different types - not simultaneously - but alternately. Two examples are indicative of this need.

First, consider an array that contains pointers to different types of records. This is shown graphically in figure 4.11.



ARRAY OF UNION OF POINTERS - FIGURE 4.11

In figure 4.11 the array elements are pointers, some of which point to R1-type records and some to R2-type records. To accomplish this, we must declare the array elements as UNION type. In this way, the array elements can be either union member type (i.e. a pointer to either type of record).

This concept is expressed in SWL syntax below.

```
TYPE
  R1 = RECORD
    A,B,C:INTEGER,
    RECEND,
  R2 = RECORD
    W,X,Y,Z:BOOLEAN,
    RECEND,

  UN = UNION (AR1,AR2),

  AR = ARRAY [1..10] OF UN;

VAR
  PTRARY : AR;
```

UNION SYNTAX - FIGURE 4.12

Another example uses the concept of union to allow the declaration of a procedure (or function) that can accept as an argument any one of several types.

Unfortunately, at the present time only union of pointer types is allowed. But one day, union of other types will also be available. So in our example below we have a procedure which will free a linked list. This is an extension and generalization of the example in figure 3.33. The procedure will accept as an argument one of many types of pointer to linked list. We will show only the procedure declaration at this time.

```
TYPE
  R1 = RECORD
    "R1 RECORD DECLARATION"
  RECEND,
  R2 = RECORD
    "R2 RECORD DECLARATION"
  RECEND,
  R3 = RECORD
    "R3 RECORD DECLARATION"
  RECEND,
  UN = UNION (^R1,^R2,^R3);
:
PROC FREELIST (VAL PTR:UN);
:
PROCEND FREELIST;
```

#### UNIONS IN PROCEDURES - FIGURE 4.13

In figure 4.13 the union type UN can contain a pointer to one of R1, R2 or R3 type records. The procedure FREELIST can accept as an actual parameter either of the three types of pointers. The contents of the procedure FREELIST will be discussed below.

#### UNION OPERATORS

There are three UNION operators. These are:

OPERATOR	MEANING
::	TYPE TESTING
::=	VALUE TYPE TESTING
::^	POINTER TYPE TESTING

#### UNION OPERATORS - FIGURE 4.14

Of the three union operators illustrated in figure 4.14 only the value type testing :=: operator is currently available. Therefore, we will discuss this operator only.

#### VALUE TYPE TESTING

The value type testing operator performs two operations. First, if the variable on the left is the same type as the current value of the union variable on the right the operator returns the boolean value TRUE; otherwise, the boolean value FALSE is returned. Second, when the boolean value is true then the variable on the left is assigned the current value of the union variable on the right. We can test to determine what type is currently in the variable and obtain the contents of the union variable.

Keeping in mind that only unions of pointers are allowed at the present time we can finish the contents of the FREELIST procedure that we started in figure 4.13.

```
"ASSUME RECORD TYPES DECLARED"  
"AS IN FIGURE 4.13      "  
  
TYPE  
  UN = UNION (A R1, A R2, A R3);  
  
PROC FREELIST (VAL PTR:UN);  
VAR  PTR1:A R1,  
     PTR2:A R2,  
     PTR3:A R3;  
  
IF PTR1:=: PTR  
  THEN "CODE TO FREE R1 TYPE LIST"  
  ORIF PTR2:=: PTR  
  THEN "CODE TO FREE R2 TYPE LIST"  
  ORIF PTR3:=: PTR  
  THEN "CODE TO FREE R3 TYPE LIST"  
  ELSE "ERROR"  
IFEND;  
  
PROCEND FREELIST;
```

#### VALUE TYPE TESTING OPERATOR - FIGURE 4.15

In the example in figure 4.15, the formal parameter "PTR" is a union of three possible pointers to three different records. The variable declaration identifies three variables each one is a pointer to a different type record.

In the statement "IF PTR1 :=: PTR", the value type test operator :=: tests the type of PTR1 (pointer to R1 type record list) against the current type of the contents of the union variable PTR.

If the types are not identical, then the Boolean result FALSE is returned and the IF statement progresses to the next ORIF clause. If the types match, then the boolean value of `:=:` is TRUE and the variable PTR1 is set (or assigned) to the current value of the union variable. When the result of the value testing operator `:=:` is true, the THEN clause of the IF statement is executed and the code to free the list is executed.

You can see how much effort must be used to isolate and use the current contents of the union variable.

#### VALUE CONFORMITY CASE

The value conformity CASE statement provides a method of clearly identifying the case to be performed depending upon the current value of a union variable. The value conformity case, then, is an extension of the IF and ORIF clauses (in figure 4.15) necessary to isolate the current value of a union and execute the appropriate statements.

Lets rewrite the FREELIST procedure of figure 4.15 using the value conformity CASE statement.

```

"ASSUME RECORD TYPES DECLARED"
"AS IN FIGURE 4.13          "

TYPE UN = UNION (^R1, ^R2, ^R3);
PROC FREELIST (VAL PTR:UN);
VAR PTR1 :^R1,
    PTR2 :^R2,
    PTR3 :^R3;

CASE :=: PTR OF
=PTR1=  "CODE TO FREE R1 TYPE LIST"
=PTR2=  "CODE TO FREE R2 TYPE LIST"
=PTR3=  "CODE TO FREE R3 TYPE LIST"
ELSE    "ERROR"
CASEND;

PROCEND  FREELIST;

```

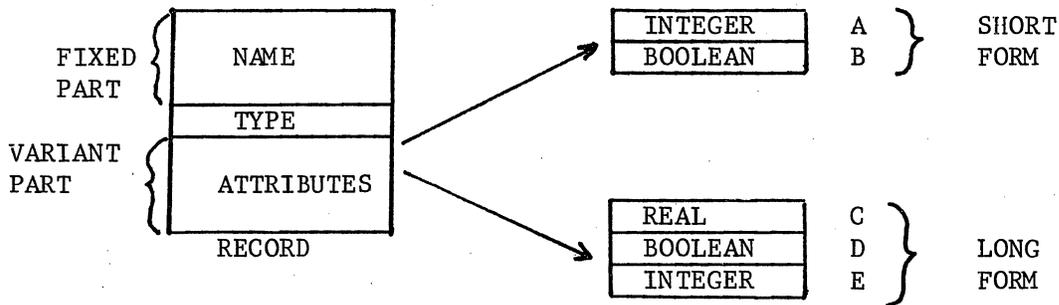
#### VALUE CONFORMITY CASE SYNTAX - FIGURE 4.16

You can see how this value conformity CASE statement makes the program text clearer. The first line of the CASE statement "CASE :=: PTR OF" clearly identifies the statement as testing the value (`:=:`) of the union variable PTR. The possible cases (`=PTR1=` etc.) clearly indicate what the alternatives are. With the conformity case statement, we do not need to bother with the TRUE-FALSE values normally associated with the type testing operators.

When the type of a case (i.e. `=PTR1=`) does match the current type of the union variable (PTR) the variable in the case (PTR1) is assigned the current value of the union. This value (PTR1) can then be used in the executable code of the conformity case statement.

## VARIANT RECORDS

When we are defining tables for system software, we often require a record with some fixed information (or fields) followed by some variable (variant) fields. Usually, we know what all the variants look like. Figure 4.17 shows this pictorially.



VARIANT RECORD CONCEPT - FIGURE 4.17

Figure 4.17 shows a record consisting of two major parts: 1) the name and 2) some attributes. Depending upon the type of the record the attributes portion of the record can be one of two forms (the short form or the long form).

In any one occurrence of the record, only one of the two forms will exist. However, some occurrences of the record may be the short form and others may be the long form.

This kind of record structure can be declared in SWL.

```
TYPE
  FORM = (SHORT, LONG),

  RECTYPE = RECORD
    NAME: STRING (11) OF CHAR
    CASE T:FORM OF
      =SHORT= A: INTEGER,
              B: BOOLEAN,
      =LONG=  C: REAL,
              D: BOOLEAN,
              E: INTEGER,
    CASEEND
  RECEND;

  VAR  ONEREC, TWOREC: RECTYPE;
```

SWL VARIANT RECORD SYNTAX - FIGURE 4.18



## LABELS

In the current ISWL (not SWL) language all labels must be declared with the LABEL statement. The label statement simply declares its constituent identifiers as labels.

Labels are often required by the EXIT and CYCLE statements to identify which statement is being EXITed or CYCLED. Labels are also used in the GOTO statement.

A labeled statement is one that includes a label followed by a colon preceding the statement itself.

```

    LABEL FORLOOP, LOOP1;
    .
    .
    LOOP1: LOOP
        .
        .
        LOOPEND
    .
    .
    FORLOOP:
    FOR
    .
    .
    FOREND;

```

] LOOP1

] FORLOOP

LABELS - FIGURE 4.21

Notice, in figure 4.21, that the label may be placed to the left of a statement (as in LOOP1) or on a line by itself (as in FORLOOP). In either method, the label identifies the whole statement. The label LOOP1 refers to the entire LOOP-LOOPEND statement. A GOTO statement referencing LOOP1 would initialize (or begin execution) at the first constituent statement of the LOOP statement. An EXIT statement, inside the FOR statement, that EXITs FORLOOP will continue execution (branch to) at the statement after FOREND.

## EXIT REVISITED

The EXIT statement was discussed in Chapter 2. Recall that the EXIT statement exits the structured statement in which it is contained. By using a label with the EXIT statement it is possible to specify which statically encompassing structured statement is to be EXITed. Consider the program skeleton shown in figure 4.22.

```

WHILE A<B DO
.
.
IF C=D
    THEN EXIT
    ELSE "SOME STATEMENTS"
IFEND;
.
.
WHILEND;

```

NON-LABELED EXIT - FIGURE 4.22

In the example in figure 4.22 the EXIT statement exits the structured IF statement.

Is it possible for the EXIT statement to EXIT the WHILE statement; that is, to cause control to be transferred to the statement after WHILEND? The answer is yes, using a label. See figure 4.23.

```

L:EXIT L;
.
.
L:WHILE A<B DO
.
.
IF C=D
    THEN EXIT L
    ELSE "SOME STATEMENTS"
IFEND;
.
.
WHILEND;

```

LABELED EXIT - FIGURE 4.23

In the example in figure 4.23, when the "EXIT L" statement is executed control will be transferred to the statement after the one labeled L. This statement is of course the one after the WHILEND because label L designates the entire WHILE statement (from WHILE to WHILEND).

Notice that the EXIT statement is a kind of restricted GOTO. It does not provide the ability to arbitrarily go anywhere in the program (the GOTO doesn't either as we will find out). The EXIT statement with a label allows the orderly exit from any statement (which is labeled) that statically (in the source code) encompasses the EXIT statement.

## CYCLE STATEMENT

The CYCLE statement allows the conditional bypassing of the statements inside a repetitive statement (FOR, LOOP, WHILE, REPEAT) and cycles to the next iteration, if any.

The CYCLE statement may use a label to designate which repetitive statement is being cycled. The CYCLE statement can only cycle repetitive statement that statically encompass the CYCLE statement.

The CYCLE statement can also be made conditional by using the WHEN clause. Examples of non-labeled CYCLE statements are given in figure 4.24.

```
FOR I:= 1 TO 10 DO
    CYCLE WHEN I = 6;
    .
    WRITE (I,EOL);
    .
    .
FOREND;

WHILE C/=EOL DO
    .
    .
    IF C = A [ I ] THEN CYCLE;
    .
    .
    READ (C);
WHILEND;
```

### NON-LABELED CYCLE STATEMENT - FIGURE 4.24

In figure 4.24 the FOR statement would create 10 iterations. However, when the CYCLE statement is executed and I=5 the statements from CYCLE to FOREND will be skipped and the next iteration (I:=7) will be done. The WRITE statement then would write the values of I: 1,2,3,4,5,7,8,9,10.

The WHILE statement in figure 4.24 shows how the simple CYCLE statement can be used to cycle a repetitive statement (e.g., the WHILE statement).

```

        LABEL  L1,L2
        .
        .
L1:REPEAT
        .
        .
L2:WHILE
        .
        .
        FOR I:= 1 TO 10 DO
        .
        .
        CYCLE WHEN A = B [I];
        CYCLE L2 WHEN A<B [I];
        CYCLE L1 WHEN A>B [I]
        FOREND;
        WHILEND;
        UNTIL C=EOL;

```

LABELLED CYCLE STATEMENT - FIGURE 4.25

Figure 4.25 illustrates the use of labels in the CYCLE statement. The statement "CYCLE WHEN A=B [I];" will cause the FOR statement to be cycled when B [I] = A. The statement "CYCLE L2 WHEN A<B [I];" will cause the WHILE statement (statement L2) to be cycled. The statement "CYCLE L1 WHEN A>B [I]" will cause the REPEAT statement to be cycled.

### RETURN

The RETURN statement will cause control to be transferred (back) to the procedure that called the procedure containing the return statement.

At execution time, a return is automatically done when we reach the end of a procedure or function (PROCEND). For this reason we have not needed the return statement for our examples thus far.

But, what if we wish to return from some place other than the end of the procedure or function? Then the RETURN statement is needed. The RETURN statement can also have (optionally) a WHEN clause making the RETURN statement conditional.

Figure 4.26 illustrates the use of the RETURN statement in a function.

```
PROC CHECK (VAL M:INTEGER) BOOLEAN;  
  
  IF M>=0  
    THEN CHECK:=TRUE;  
         RETURN;  
    ELSE CHECK:= FALSE;  
         RETURN;  
  IFEND;  
  
PROCEND CHECK;
```

#### USE OF RETURN STATEMENT - FIGURE 4.26

In figure 4.26, the function CHECK returns the boolean value TRUE when the parameter passed is positive or zero and returns the boolean value FALSE when the parameter passed is negative.

#### GOTO STATEMENT

The GOTO statement is not our favorite statement. In fact, we try to avoid writing GOTO's. But, if you insist - read on.

The GOTO statement causes transfer of control to the statement designated by a label.

If the label referenced is outside the current block then the form "GOTO EXIT label" must be used.

#### ADVANCED INPUT/OUTPUT

In Chapter 2, we discussed the simple I/O procedures READ and WRITE. These procedures provided the capability to read the input file and write on the output file. There were certain types that could be written. The input and output files were opened automatically at job initiation and closed automatically at job termination.

Now we want to examine a more complicated world of I/O. In this environment we must open and close files explicitly and we must manipulate the pointer to the file. We can handle either binary or coded files.

The I/O statements discussed here are: GET, PUT, REWIND, REWRITE, OPEN, CLOSE, #EOF, and the file type.

Before we can do any advanced I/O, we must declare the filename, type of file, and intended use of the file.

```
TYPE
  REC = RECORD
    NAME : STRING (10) OF CHAR,
    DAY : 1..31,
    MONTH : 1..12,
    YEAR : 1975..2000,
    INFO : INTEGER,
    RECEND;
```

```
VAR
  F1 [IN] : FILE OF CHAR,
  F2 [OUT] : FILE OF INTEGER,
  F3 [IN] : FILE OF REC;
```

#### FILE DECLARATIONS - FIGURE 4.27

In Figure 4.27, we see examples of file declarations. The file named F1 is an input file of characters. Each GET operation will access one character. Character files are coded files.

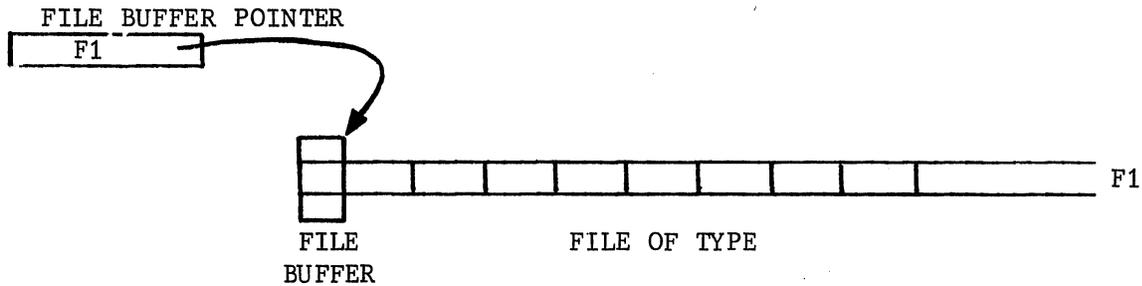
F2 and F3 are binary files. File F3 illustrates how a file of some arbitrary type can be declared. For file F3 each GET or PUT will access one record. That is, the file declaration defines the unit of information transferred with one I/O statement.

The names F1, F2 and F3 function in two capacities. First, they are the name of the file. Second, they are the pointers to the elements in the file.

#### OPEN

Before we can perform any advanced operations on a file the file must be explicitly opened (files INPUT and OUTPUT, however, are opened automatically). The OPEN statement opens the file specified and assigns the first component to the file buffer pointer.

An illustration of the file structure is shown in figure 4.28.



```

VAR
  F1 [ IN ] : FILE OF REC,
  X : REC;
.
.
OPEN (F1);
X:=F1^;
.
.

```

FILE STRUCTURE - FIGURE 4.28

Figure 4.28 illustrates access to a file. The file F1 is declared to be a file of REC (from figure 4.27). That means each component on the file is a record. Notice that a variable X is also declared. The "OPEN (F1);" statement opens the file F1 and positions the file buffer over the first component and sets the file buffer pointer to point to the file buffer. The statement "X:=F1^;" assigns to X what the file buffer pointer points to (i.e. the first component in the file). In this manner, we have just "read" the first record from the file.

GET STATEMENT

To continue our reading process, we must move the file buffer to the next component and update the file buffer pointer accordingly. These operations are accomplished by the GET statement. Note that GET does not retrieve data. GET positions the file buffer and the associated file buffer pointer. To read data, we always use the syntax filename ^.

Now we can write a program skeleton to open a file and read the components, effectively transferring the components from a file to an array in memory.

```

"ASSUME RECORD DECLARATION IN FIGURE 4.27"
CONST LIMIT = 100;
VAR F1 [ IN ] : FILE OF REC,
    X : ARRAY [ 1..LIMIT ] OF REC,
    I : [ STATIC ] 1..LIMIT + 1 := 1;
OPEN (F1);
WHILE NOT #EOF (F1) AND (I<=LIMIT) DO
  X [ I ] := F1^;
  I := I+1;
  GET (F1);
WHILEND;
WRITE ('END OF READ', EOL);
GET-FILE I/O - FIGURE 4.29

```

## PUT STATEMENT

The PUT statement appends the value of the file buffer variable (f<sub>Λ</sub>) to the file and then the file buffer variable (f<sub>Λ</sub>) becomes undefined.

For example, let's consider writing the contents of an array of records onto a file for use later.

```
"ASSUME RECORD DECLARATION AS IN FIG. 4.27"  
CONST LIMIT = 100  
VAR X: ARRAY [1..LIMIT] OF REC,  
      I: 1..LIMIT+1,  
      F2 [OUT]: FILE OF REC;  
"ASSUME ARRAY X HAS BEEN INITIALIZED"  
OPEN (F2);  
FOR I:= 1 TO LIMIT DO  
  F2 ^ := X [I];  
  PUT (F2);  
FOREND;  
.  
.
```

PUT - FILE - Figure 4.30

## OTHER I/O STATEMENTS

The REWIND statement is used to reset the current position to the beginning of the file and assigns the file buffer pointer to the first component in the file. To rewind file F1, the statement "REWIND (F1);" would be used.

The CLOSE statement closes the file. I/O operations are not allowed in the file once it has been closed. To close file F1, the statement "CLOSE(F1);" would be used.

## DATA-REPRESENTATION DEPENDENT FEATURES

The facilities covered in this section can be used only in a procedure or function which is declared [REPDEP]. Generally, you will not need (and will not use) these facilities. Occasionally, you will need these features, and when you do they MUST be declared [REPDEP].

One facility, which is needed occasionally, is the ability to measure a variable's storage space requirements in the addressable unit of memory for the machine on which the SWL program is executing. This kind of ability would certainly be required in a memory management utility for example.

To support the need for assigning values to the unit of addressable storage, the CELL data type was created.

Only assignment and equality tests are defined on a CELL.

To manipulate cells two functions are provided:

- 1) #SIZE (variable identifier) function returns the number of CELLS required to contain the variable identifier or type identifier specified;
- 2) #LOC (argument) returns a pointer which can be assigned to any direct pointer. These functions, when used, must appear in a [REPDEP] procedure or function.

For example, the following function returns an integer indicating the number of cells required to contain the specified argument (which is an array in this example).

```
TYPE A =ARRAY [*] OF INTEGER
PROC [REPDEP] MEMORY
  (VAL M:A ;) INTEGER;
MEMORY := #SIZE (M);
PROCEND MEMORY;
```

REPDEP FUNCTION #SIZE - Figure 4.31

The next example illustrates how values may be assigned to CELL type.

```
PROC [XDCL REPDEP] MAIN;
VAR A,C : CELL,
    P : ^INTEGER;

P:= #LOC (A)
P^:= 0110(8);
C := A;
.
.
PROCEND MAIN;
```

REPDEP CELL AND #LOC FUNCTION - Figure 4.32

Many interesting things are happening in figure 4.32.

Notice the statement "P:= #LOC (A);". P is a pointer to an integer. A is type cell. #LOC (A) returns a pointer to type cell. This assignment then is quasi-legal. That is, it assigns a pointer to a cell to a variable that is a pointer to an integer. This is the mixing of types that requires the REPDEP attribute. #LOC will provide a pointer to its argument that can be assigned to any pointer variable.

✓ The next statement "P^:= 0110(8);" is no surprise. It simply assigns the value 0110(8) to what P points to (the cell). Unfortunately, this is the only way to assign constants directly to a cell.

Once cells have been initialized, we can make assignments as shown in the statement "C:=A;". Here, we are simply assigning the value of cell A to cell C (no type mixing here).

## STANDARD FUNCTIONS

We have covered most of the standard functions available in the language throughout the text. Below you will find a list of the current standard functions.

<u>FUNCTION</u>	<u>REFER TO</u>
\$REAL (X)	CHAPTER 2, fig 2.11
\$INTEGER (X)	CHAPTER 2, fig 2.11
\$CHAR (X)	CHAPTER 2, fig 2.11
\$STRING (1,s,f)	CHAPTER 2
#STRLENGTH (S)	CHAPTER 4, fig 4.10
#LOWERBOUND (array,N)	CHAPTER 4, fig 4.8
#UPPERBOUND (array,N)	CHAPTER 4, fig 4.8
#EOF (file name)	CHAPTER 2, fig 2.25
#LOC (argument)	CHAPTER 4, fig 4.32
#SIZE (argument)	CHAPTER 4, fig 4.31
#STRINGREP (VAL, SUBSTR, WIDTH, DECIMALS)	CHAPTER 2
#SUCC (X)	
#PRED (X)	
#ABS (X)	

STANDARD FUNCTIONS - Figure 4.33

## SUCCESSOR & PREDECESSOR FUNCTIONS

The #SUCC (argument) function returns the successor of its argument. For example #SUCC (3) is the value 4 because the successor of three is four.

If an ordinal was declared:

```
TYPE
    COLOR = (RED, YELLOW, BLUE, GREEN);
```

Then #SUCC (YELLOW) would be BLUE. The #SUCC ('A') is 'B', etc.

The #PRED (argument) function is similar; it returns the predecessor of its argument.

What happens if we request a predecessor or successor that does not exist (like #SUCC (GREEN) or #PRED (RED))? The result becomes undefined!

## COMPILE TIME OPTIONS<sup>1</sup>

Compile time options are currently implemented through the mechanism of comment toggles. A comment toggle is a toggle that is inside a comment line (anywhere in the program).

Through the use of "comment toggles", the programmer has control over certain aspects of the generated code and the source listing. A toggle is a switch associated with a particular feature (such as checking for zero-divide), and it may be turned on or off by means of a comment toggle list. A comment toggle list may appear anywhere within a comment, and its syntax is:

```
<comment toggle list> ::= $<toggle> , <toggle>
toggle ::= A<plus or minus> "assignment checking"
          X<plus or minus> "index checking"
          D<plus or minus> "zero-divide checking"
          R<plus or minus> "rounding arithmetic"
          T<plus or minus> "sets all of A, X, D, and R"
          C<plus or minus> "list object code"
          L<plus or minus> "listing on or off"
                        (lines with errors listed
                          even if $L-)
          S<integer> "skip<integer> lines on
                    listing"
          S "skip to new page on listing"
          K<plus or minus> "overprint SWL reserved words"
          M<integer> "terminate scan of source
                    lines after column <integer>"
          B<integer> "start scan of source lines on
                    column <integer>"
```

<plus or minus> ::= +|-

Embedded blanks may not appear within a comment toggle list.

So, for example, to cause overprint of SWL reserved words a comment toggle would be constructed: "\$K+". To turn off the overprint of keywords later on in the program, "\$K-" could be used.

The compiler initially sets the toggles to correspond to

```
"$T+,C-,L+,K-,M72,B1"
```

which implies that all run-time checking is activated; arithmetic is performed with rounding instructions; a source listing is produced but without object code listing and overprinting; and the source lines are scanned from column 1 up to and including column 72.

1 - Compile time options described here are for ISWL only.

An entirely different mechanism of specifying compile time options will be used in SWL.

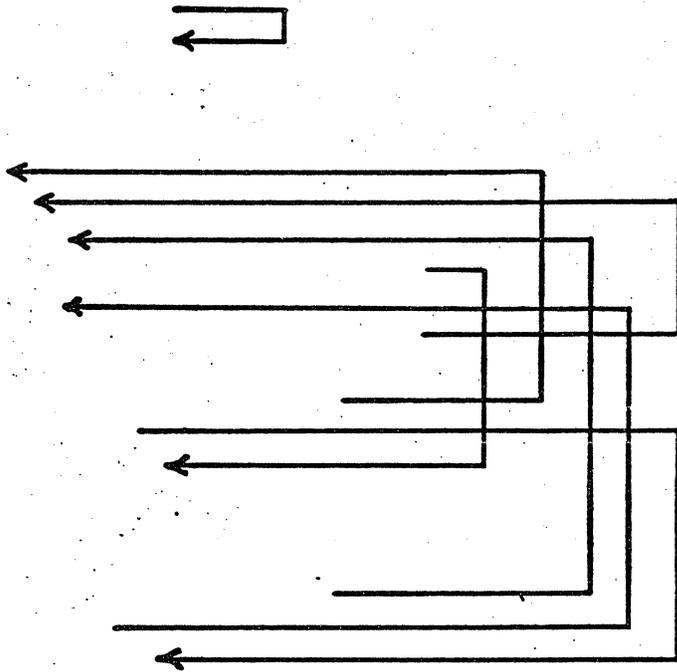
## CHAPTER 5

### STRUCTURED PROGRAMMING AND SWL

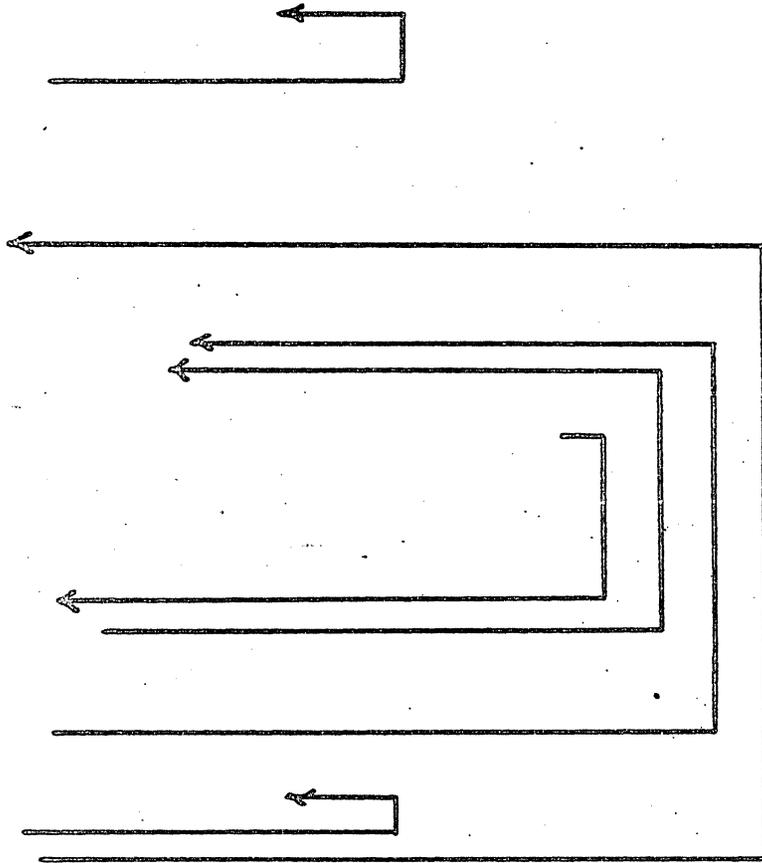
One of the primary goals we hope to attain with the use of SWL is the reduction in cost of developing software products. This reduction, hopefully, will take place to some extent in the implementation phase of software development and to a greater extent in the ongoing maintenance phase of the software developed.

It is hoped that a high level, block-structured language (SWL) will help make these dreams a reality. Our dream is software which is easy to understand. Hence, it will be easier to implement and easier to maintain and modify. In our visions, we see programs with a well-defined (and easy-to-understand) structure.

It is hoped that implementors will structure both their thoughts, and their programs. Not much can be done to tell people how to structure their thoughts. With respect to programs, however, we can suggest ways of structuring both data and executable lines of code.



AN UNSTRUCTURED PROGRAM - Figure 5.1



A STRUCTURED PROGRAM - Figure 5.2

Figure 5.1 illustrates graphically a sorting procedure which was written in an unstructured way (in FORTRAN). Notice how the lines of control cross and intermingle in this procedure. It would be difficult to understand this flow of control. As you can imagine the task of modifying the procedure is complicated by the many different ways we can get to many statements.

Contrast this unstructured flow of control with the nicely nested flow of control in figure 5.2. This flow of control (figure 5.2) resulted from a SWL program written to perform exactly the same sorting algorithm. Notice the elegant structure. It becomes obvious which loops are the outermost controlling loops. Making changes to this SWL program should be much easier.

Of course, it is possible to write a structured FORTRAN program or an unstructured SWL program - that's not the point. The point is that you should be aware of the structure and strive to make your program structures as clear and meaningful as possible.

This goal will be easily achieved if, when writing SWL code, you try to avoid the GOTO statement and concentrate on the structured statements including the IF, WHILE and REPEAT statements.

One approach to help make programs clearer is to develop the programs in a step-by-step manner. This concept was discussed in a paper by NIKLAUS WIRTH entitled "PROGRAM DEVELOPMENT BY STEPWISE REFINEMENT" which appeared in COMMUNICATIONS of the ACM, Volume 14, Number 4, April 1971. The program solutions were modified to SWL by John Sutherland of Control Data Canada Limited.

The program we will use to illustrate these concepts is a solution to the "8-Queens problem". The problem may be stated:

Given are an 8 x 8 chessboard and 8 queens which are hostile to each other. Find a position (or square) for each queen so that no queen may be taken by any other queen (i.e. every row, column and diagonal contains at most one queen).

Our program will find one solution to the problem. In thinking about the problem it is clear that we will want to test a square (by considering a queen to be there). If the position is safe, then we will set the queen at that position and move on to the next column.

If the position is not safe, we want to consider additional positions in the column. If we cannot find an acceptable position in a column, we must go back to the preceding column and move the queen there.

The program ends successfully if we try to move on to the 9th (non-existent) column. This of course means that we have successfully placed all 8 preceding queens.

The program ends unsuccessfully if we have to move the queen in the first column to the 8th rank unsuccessfully and then try the preceding (non-existent) column).

So we may express our game plan at the highest level in the following way (figure 5.3).

```
INITIALIZE;
REPEAT
  TESTSQUARE;
  IF SAFE
    THEN SETQUEEN;
      NEXTCOLUMN;
    ELSE ADVANCEQUEEN;
  IFEND;
UNTIL ENDOFSEARCH;
WRITERESULT;
```

### PROGRAM OVERVIEW - Figure 5.3

Our intention is reasonably clear. After some initialization, we will REPEAT until ENDOFSEARCH. The search algorithm entails first the test of a square to see if it can be occupied (TESTSQUARE). If the square is safe, then we will SETQUEEN on this square and go to the NEXTCOLUMN. If the square was not SAFE, then we ADVANCEQUEEN. ADVANCEQUEEN then must prepare for the next TESTSQUARE. We will also provide ADVANCEQUEEN with enough intelligence to determine when it cannot advance in one column and, hence, **must** backtrack to a preceding column.

Our goal is to leave the text of the program in figure 5.3 intact and arrange our data structure and procedures so that this high-level description of the program is correct. Note that this approach closely resembles the way we think about problems.

With this high-level design completed, we must turn to the data structures to make the program work. We **need to be able to identify valid and invalid positions for each queen**. There are four components to this 1) horizontal validity 2) vertical validity, 3) left diagonal validity, and 4) right diagonal validity.

We can handle horizontal and vertical validity with one array (indexed 1 to 8), making sure that there is only one queen in each row. We could call this array HORIZONTAL. HORIZONTAL[3]=FALSE indicates that the third row is not available for a queen (it has a queen in it).

The right diagonals can be represented as an array (indexed 2 to 16), where the diagonal is found by adding the row and column. For example row 3, column 5 (sum=8) is in the same diagonal as row 6 column 2 (sum=8). An array value TRUE would indicate that the corresponding right diagonal is available for a queen.

The left diagonal can be represented as an array (indexed -7..7) where the diagonal is found by subtracting the row minus the column. For example row 3, column 6 (difference = -3) is in the same left diagonal as row 2, column 5 (difference = -3). An array value TRUE would indicate that the corresponding left diagonal is available for a queen.

The position array (indexed 1 to 8) contains values indicating the row of the queen. For example POSITION [3] = 7 would indicate that the third column contains a queen in the seventh row.

Now lets examine some of the procedures. Here we step down to a lower level of refinement.

INITIALIZE must set up all arrays to their initial values.

TESTSQUARE must check to see if a queen can be placed on a given square (row and column). TESTSQUARE is shown in figure 5.4.

```
PROC TESTSQUARE;  
  SAFE := A[ROW] AND  
         B[ROW+COLUMN] AND  
         C[ROW-COLUMN];  
PROCEND TESTSQUARE;
```

TESTSQUARE PROCEDURE - Figure 5.4

The SETQUEEN procedure must mark rows, left diagonals, and right diagonals when they have been used. Also, when a position is used it must be saved in the position array.

See figure 5.5 below:

```
PROC SETQUEEN;  
  A[ROW] := FALSE;  
  B[ROW+COLUMN] := FALSE;  
  C[ROW-COLUMN] := FALSE;  
  POSITION[COLUMN] := ROW;  
PROCEND SETQUEEN;
```

SETQUEEN PROCEDURE - Figure 5.5

The remaining procedures are equally simple. A complete program listing can be found in figure 5.6.

The important point is that our programming language was able to implement, in a reasonable way, the logical expression of our problem.

```

000001 !MODULE QUEENS;
000063 !
-----
000063 !VAR HORIZONTAL : ARRAY [1..8] OF BOOLEAN,
000073 !   RDIAGNOL      : ARRAY [2..16] OF BOOLEAN,
000112 !   LDIAGNOL      : ARRAY [-7..7] OF BOOLEAN,
000131 !   POSITION       : ARRAY [1..8] OF INTEGER,
000141 !   SAFE          : BOOLEAN,
000142 !   ROW           : 1..9,
-----
000143 !   COLUMN        : 0..9,
000144 !   I              : -7..16;
000145 !
000145 !PROC REMOVEQUEEN;
000145 !REPEAT
000150 ! COLUMN := COLUMN - 1;
-----
000154 ! IF COLUMN < 1 THEN RETURN IFEND;
000156 ! ROW := POSITION[COLUMN];
000166 ! HORIZONTAL[ROW]:=TRUE;
000173 ! RDIAGNOL[ROW+COLUMN]:=TRUE;
000200 ! LDIAGNOL[ROW-COLUMN]:=TRUE;
000206 ! ROW := ROW + 1;
-----
000212 !UNTIL ROW <= 8;
000214 !PROCEND REMOVEQUEEN;
000216 !
000216 !
000216 !PROC INITIALIZE;
000216 !FOR I := 1 TO 8 DO
-----
000227 ! HORIZONTAL[I] := TRUE FOREND;
000235 !FOR I := 2 TO 16 DO
000244 ! RDIAGNOL[I] := TRUE FOREND;
000252 !FOR I := -7 TO 7 DO
000261 ! LDIAGNOL[I] := TRUE FOREND;
000267 !FOR I := 1 TO 8 DO
-----
000276 ! POSITION[I] := 0 FOREND;
000304 !COLUMN:=1;
000305 !ROW :=1;
000306 !PROCEND INITIALIZE;
000310 !
000310 !PROC WRITERESULT;
-----
000310 !IF COLUMN < 1
000313 ! THEN WRITE(' NO MORE SOLUTIONS',EOL);
000331 !   RETURN IFEND;
000332 !FOR ROW := 1 TO 8 DO
000341 !WRITE(POSITION[ROW]:5);
000346 !FOREND;
-----
000350 !WRITE(EOL);
000354 !PROCEND WRITERESULT;
000356 !
000356 !
000356 !PROC TESTSQUARE;
000356 ! SAFE := HORIZONTAL[ROW] AND
-----
000365 !   RDIAGNOL[ROW+COLUMN] AND
000372 !   LDIAGNOL[ROW-COLUMN];
000402 !PROCEND TESTSQUARE;

```

STEPWISE REFINEMENT - Figure 5.6

```

000404 !
000404 !PRØC SETQUEEN;
000404 ! HØRIZØNTAL[RØW] := FALSE;
000413 ! RDIAGNØL[RØW+CØLUMN] := FALSE;
000420 ! LDIAGNØL[RØW-CØLUMN] := FALSE;
-----
000425 ! PØSITION[CØLUMN] := RØW;
000432 !PRØCEND SETQUEEN;
000434 !
000434 !PRØC NEXTCØLUMN;
000434 ! CØLUMN := CØLUMN + 1;
000442 ! RØW := 1;
-----
000443 !PRØCEND NEXTCØLUMN;
000445 !
000445 !PRØC ADVANCEQUEEN;
000445 ! RØW := RØW + 1;
000454 ! IF RØW > 8 THEN REMØVEQUEEN; IFEND;
000457 !PRØCEND ADVANCEQUEEN;
-----
000461 !
000461 !PRØC ENDØFSEARCH BØØLEAN;
000461 ! ENDØFSEARCH:=(CØLUMN<1) ØR (CØLUMN>8);
000473 !PRØCEND ENDØFSEARCH;
000475 !
000475 !PRØC [XDCL] MAIN;
-----
000475 ! INITIALIZE;
000501 !REPEAT
000501 ! TESTSQUARE;
000502 ! IF SAFE
000502 ! THEN SETQUEEN;
000504 ! NEXTCØLUMN;
-----
000505 ! ELSE ADVANCEQUEEN;
000507 ! IFEND;
000507 !UNTIL ENDØFSEARCH ();
000511 !WRITERESULT;
000512 !PRØCEND MAIN;
000514 !MØDEND QUEENS;
-----

```

STEPWISE REFINEMENT - Figure 5.6 (Continued)

To find the complete list of solutions to this problem (all 92 of them) we need only cause the repeat sequence of procedure MAIN to be repeated as shown in figure 5.7.

```
000477 !PRØC [XDCL] MAIN;
000477 !INITIALIZE;
000503 !REPEAT
000503 !REPEAT
000503 ! TESTSQUARE;
000504 ! IF SAFE
000504 ! THEN SETQUEEN;
000506 !     NEXTCØLUMN;
000507 ! ELSE ADVANCEQUEEN;
000511 ! IFEND;
000511 !UNTIL ENDØFSEARCH ();
000513 !WRITERESULT;
000514 !IF CØLUMN >= 1 THEN ADVANCEQUEEN IFEND;
000517 !UNTIL CØLUMN < 1;
000521 !PRØCEND MAIN;
```

MODIFICATION TO PROGRAM OF FIGURE 5.6 - Figure 5.7



## CHAPTER 6

### SWL PROGRAMMING TECHNIQUES AND CONVENTIONS

This chapter is an assembly of techniques and conventions that may be used at your discretion. Unfortunately, at this time there are no conventions. There are also precious few good programming techniques. It is hoped that readers will provide some feedback and suggest additional SWL Programming TECHNIQUES.

## TECHNIQUES FOR EASE OF MAINTENANCE

Almost every program declares some variables. Typically, at some later date, it becomes necessary to add variables to the program. Consider the variable declaration of figure 6.1

```
VAR R: REAL,  
    A: ARRAY [1..10] OF BOOLEAN,  
    C: CHAR;
```

### DIFFICULT TO MAINTAIN VAR DECLARATION - Figure 6.1

In figure 6.1, if we needed (for maintenance or modification) to add another variable to the declaration (presumably at the end of the list of variables, we would have to change two lines. First, the semi-colon must be changed to a comma. Second, the new variable line must be added. The result is two modifications. Similar problems occur if we needed to delete the variable R or C from the declaration.

Now refer to the variable declaration of figure 6.2.

```
VAR  
  R: REAL,  
  A: ARRAY [1..10] OF BOOLEAN,  
  C: CHAR,  
  ;
```

### EASY TO MAINTAIN VAR DECLARATION - Figure 6.2

In figure 6.2, it is easy to add a new variable with only one modification. Similarly, deleting any variable from the list requires only one modification. This feature is not available in the current version of ISWL. Future versions will, however, support this feature.

## TECHNIQUES FOR PROGRAM CLARITY

Often, poor programming habits will cause a decrease in program clarity. One example of this is the unnecessary use of control statements (GOTO, EXIT, etc.).

Figure 6.3 illustrates two programs. The results are identical, only the source code is different. The programs check for an empty line (from a terminal). If the line is empty, a REWIND is executed and another attempt is made to find data on the line. When reading data, it is necessary to check for end-of-file after each read.

<pre> LABEL   Z,   ; VAR   SUM : INTEGER,   N   : INTEGER,   ;   "CHECK FOR EOF"   Z: IF #EOF (INPUT)     THEN REWIND (INPUT);     GOTO Z   IFEND;    "SUM UP DATA"   SUM :=0;   LOOP   READ (N);   EXIT WHEN #EOF (INPUT);   SUM := SUM+N   LOOPEND; </pre>	<pre> VAR   SUM : INTEGER,   N   : INTEGER,   ;   "CHECK FOR EOF"   WHILE #EOF (INPUT) DO   REWIND (INPUT);   WHILEND;    "SUM UP DATA"   SUM :=0;   READ (N);   WHILE NOT #EOF (INPUT) DO   SUM := SUM+N   READ (N);   WHILEND; </pre>
--	---

PROGRAM CLARITY - Figure 6.3

In figure 6.3 both examples appear similar. However, upon detailed analysis the left program requires the reader to trace lines of control created by the GOTO and EXIT statement. The right example can be read more easily from top to bottom.



## CHAPTER 7

### PERFORMANCE MEASUREMENT & PREDICTION

In this chapter, we will be discussing techniques for improving the execution time performance of your SWL programs. Since the performance tools we will be discussing exist on the CDC CYBER 70/170 computer systems under the KRONOS 2.1 operating system, the techniques are necessarily machine dependent. However, it is most likely that some similar tools will exist on future computer systems. Once you see how easy the tools are to use and how much information can be obtained from a performance study, you will certainly want to include performance testing in your software development plans.

## PERFORMANCE PREDICTION

Performance prediction is done prior to coding - often during the design stage(s) of software development. In performance prediction, we ask questions, such as:

1. Which data structure will provide most rapid access (least CPU time)?
2. Which algorithm will execute in the least amount of CPU time?
3. What disc loading will this program create during execution?

Notice that performance prediction tries to answer questions about what will happen when the software is written. Performance prediction helps us choose the best methods of software design and implementation prior to coding.

Unfortunately, at this time there exists almost no performance prediction information for SWL, so we must leave this important performance area without providing any acceptable answers. We mention the topic of performance prediction to indicate the lack of information and to put performance measurement in its proper perspective.

## PERFORMANCE MEASUREMENT

Performance measurement is done after a program is written. The performance measurement tools will enable us to find the specific statements in our SWL program that account for the greatest percent (or amount) of CPU time. Armed with this information we can make modifications to our software to make it perform faster. In addition, after making some changes we can determine just what improvement in execution time was actually realized. In all these endeavours, however, the performance considerations are done after the software has been written.

Fortunately, we have available some very good performance measurement tools. These are standard tools available to any program (FORTRAN, APL, COBOL, etc.) that runs under KRONOS 2.1 on the CDC CYBER 70/170 computer systems. These tools have been interfaced with SWL to make them easy to use.

## HOW DOES PERFORMANCE MEASUREMENT WORK?

The following discussion explains the performance measurement tools on the CDC CYBER 70/170 computer systems.

There are two distinct parts to our performance measurement tool. The first part gathers information (statistically) from the running SWL program.

This data gathering is done at a frequency of once every 100 micro seconds. The data gathering is done by a Peripheral Processing Unit (PPU) running a data collection program called SMP! SMP reads the central processor P register each  $100 \times 10^{-6}$  seconds, categorizes the location of the P register (was the P register running our program or someone else's program?), and adds a count to the appropriate location in a table in the PPU's memory. At job termination, the PPU writes the raw data (table contents) onto a file named PSAMPLE. This concludes the data gathering portion of our performance measurement tool.

The second part of our performance tool is the data reduction program. This is a standard (KRONOS supplied) FORTRAN program (named PSAMP) that reads the contents of the file named PSAMPLE and produces some nice looking data for our interpretation. This concludes the data reduction portion of performance measurement.

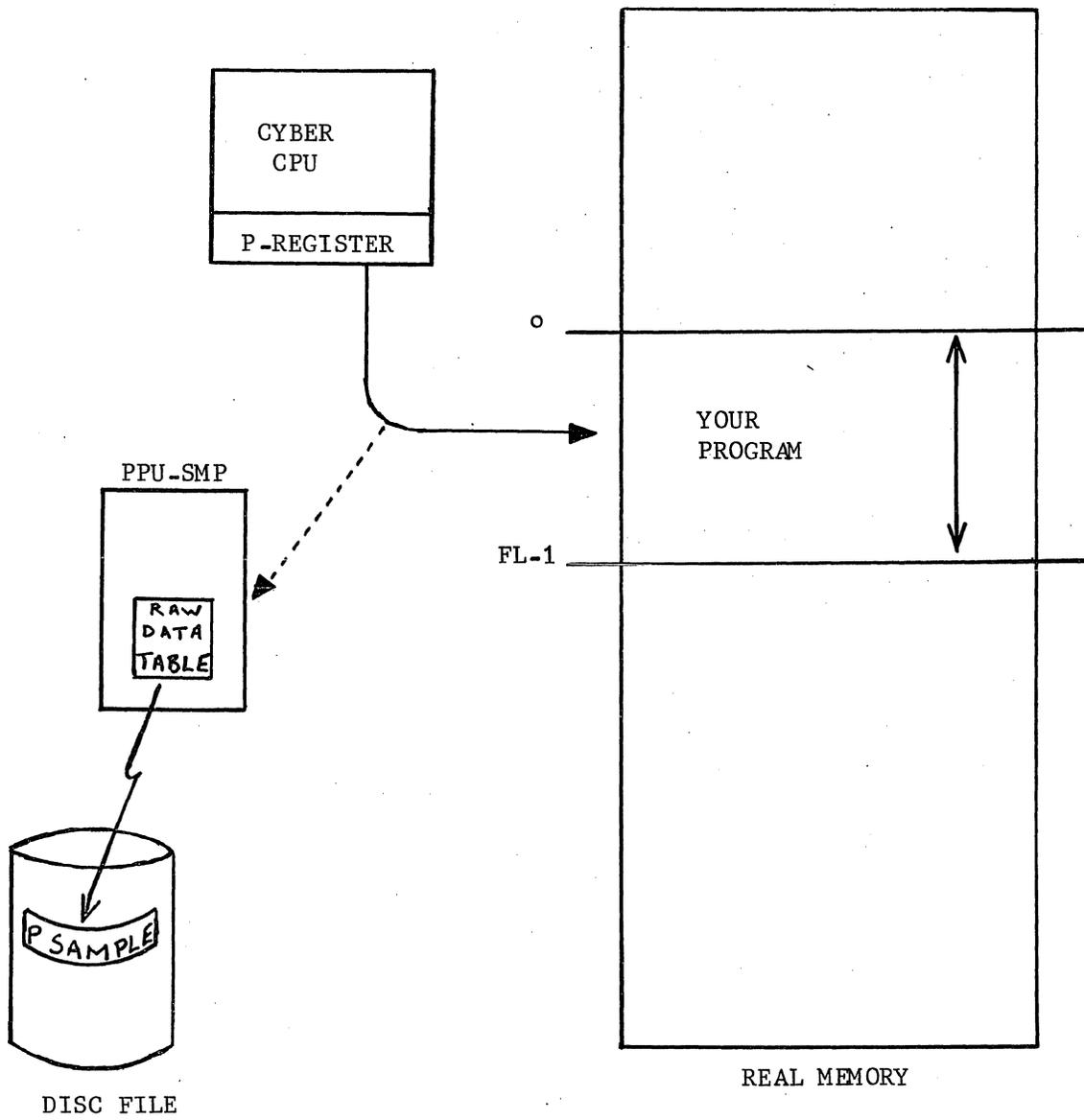
Now, our work begins. We correlate the performance information produced by the FORTRAN program (PSAMP) with our SWL source language listing. From this correlation, we can determine how much time each SWL statement (or group of statements) requires. We can also determine how much time is spent waiting for I/O and how much time is being spent in the system routines.

We may in fact, run the performance measurement tool a number of times. First we will be looking for an overview at the procedure level. We will be trying to answer the question: "which procedures require enough CPU time to warrant further investigation?". Then we may narrow the range of our performance measurement to a single procedure (or part of a procedure) to answer the question: "which SWL statements in this procedure take up most of the CPU time?".

#### SYSTEM DESCRIPTION

Figure 7.1 illustrates the important pieces of a performance measurement run. Your program resides in a portion of real memory. When a performance run is in progress your job cannot be rolled out. This implies that performance measurement runs can only be made in the background (or batch mode). There are two ways to run batch jobs: 1) read the job in from the CARD reader, or 2) use the SUBMIT command from the terminal. The only restriction is that the performance measurement job may not run interactively from the terminal. As your job is running the dedicated PPU (SMP) is sampling the P register every 100 micro seconds and updating the PPU raw data table. At job termination the file PSAMPLE is created.

The PPU raw data table is  $202_8$  entries long. One entry is for all samples found below the range you have asked to sample. A second entry is devoted to all samples above the range you have asked to see. The remaining  $200_8$  entries are for the range you have asked to examine.



PERFORMANCE MEASUREMENT SYSTEM DESCRIPTION - Figure 7.1

If, for example, you ask to examine the 200<sub>8</sub> word area in your program from 11306<sub>8</sub> to 11505<sub>8</sub>, then the raw data table will have one entry for each word in your program in the range you wanted sampled.

Remember that the PPU raw data table is fixed in length (200<sub>8</sub> entries). So, if you ask for a sample range of 300<sub>8</sub> words (say from 16214<sub>8</sub> to 16513<sub>8</sub>) then the PPU raw data table will use one entry to represent two words from your program. Hence the resolution of the sample will be (every) two words.

Figure 7.2 shows the correspondence between the range you ask to sample and the resolution of the data produced.

SAMPLE RANGE REQUESTED (R)	RAW DATA TABLE RESOLUTION
$0 < R \leq 200_8$	1 WORD
$201_8 \leq R \leq 400_8$	2 WORDS
$401_8 \leq R \leq 1000_8$	4 WORDS
$1001_8 \leq R \leq 2000_8$	10 <sub>8</sub> WORDS
$2001_8 \leq R \leq 4000_8$	20 <sub>8</sub> WORDS
$4001_8 \leq R \leq 10000_8$	40 <sub>8</sub> WORDS
$10001_8 \leq R \leq 20000_8$	100 <sub>8</sub> WORDS
$20001_8 \leq R \leq 40000_8$	200 <sub>8</sub> WORDS
$40001_8 \leq R \leq 100000_8$	400 <sub>8</sub> WORDS
$100001_8 \leq R \leq 200000_8$	1000 <sub>8</sub> WORDS

SAMPLE RANGE vs RESOLUTION - Figure 7.2

### SWL INTERFACE

The interface function is written in assembly language (COMPASS) to conform to and be callable from any SWL procedure. The interface procedure accepts two parameters (FIRSTword address to be sampled and LASTword address to be sampled) and returns a boolean result. The boolean result is always TRUE on the CDC CYBER computer system. It would be FALSE on a computer system that did not have the performance measurement tools. The COMPASS interface procedure calls the PPU program SMP and insures that only one performance measurement is allowed to run. This interlock (allowing only one sampling run) is done to insure that the computer system resources (central memory and Peripheral Processors) are not all tied up doing sampling runs.

The COMPASS interface function is equivalent to the following SWL procedure declaration:

```
PROC [XDCL] SWLSMP (VAL FWA,LWA:INTEGER) BOOLEAN;  
.  
PROCEND SWLSMP;
```

So, when you want to do a sampling run you must declare SWLSMP to be XREF as follows:

```
PROC [XREF] SWLSMP (VAL FWA,LWA:INTEGER) BOOLEAN;
```

Then, when you want to begin the sampling run the SWLSMP function is called giving the FWA and LWA to be sampled. A typical calling sequence might be:

```
IF SWLSMP (500(8),1627(8))  
  THEN WRITE ('SWLSMP OUTPUT FOLLOWS',EOL);  
  ELSE WRITE ('SWLSMP NOT AVAILABLE',EOL);  
IFEND;
```

The COMPASS interface function (SWLSMP) is available in source form and may be obtained with the command:

```
GET, SWLSMP/UN=ZED
```

SWLSMP is also reproduced in figure 7.3. Since this is not a guide for COMPASS programmers SWLSMP will not be explained in detail.

```

IDENT SWLSMP
ENTRY SWLSMP

```

```

*****
*
* WRITTEN BY : RON ROTHSTEIN *
* CDC FACILITY : TTOFAC *
* COMPOSED ON : 25 OCTOBER 1974 *
* LAST MODIFIED: 28 OCTOBER 1974 *
*
* ISWL EQUIVALENT STATEMENT: *
*
* PROC [XDCL] SWLSMP *
* (VAL FWA,LWA:INTEGER) BOOLEAN; *
*
*****

```

```

SWLSMP  SX0  B5 *****
        LX0  22B *
        BX7  X7+X0 * ENTRY SEQUENCE
        SB5  B6 *
        SA7  B5+1 *
        SB6  B6+6 *****
        SB2  XXXXXX+10 *****
        SA2  B0+B2 * INITIALIZE FET
        MX0  42D * ERROR RETURN ADDR
        BX2  X2*X0 * SO PSAMPLE BUSY
        SX3  B0+ERAD * MESSAGE WILL NOT
        BX6  X2+X3 * APPEAR IN DAYFILE
        SA6  B0+B2 *****

```

```

ATTCH  BSS  0
ATTACH XXXXXX,.,.,W
SA1  XXXXXX READ RETURN CODE
MX0  8 CONSTRUCT MASK
LX0  18 POSITION TO BITS 17-10
BX1  X0*X1 EXTRACT CODE BITS
SB1  X1 MOVE CODE TO B1

```

```

SB2  4000B PSAMPLE FILE
NE  B1,B2,NEXT NOT FOUND
SX6  B0
SA6  B5+2 PSAMPLE := FALSE;
MESSAGE MESS1,3,R PSAMPLE FILE NOT FOUND
EQ  B0,B0,EXIT RETURN

```

```

NEXT  SB2  2000B PSAMPLE FILE
      NE  B1,B2,NEXT1 FOUND BUSY
      RECALL
      EQ  B0,B0,ATTCH TRY AGAIN LATER

```

SWLSMP INTERFACE FUNCTION - Figure 7.3

NEXT1	EQ	B1,B0,CALSMP	OTHER ERRORS
	SX6	B0	
	SA6	B5+2	PSAMPLE := FALSE;
	MESSAGE	MESS3,3,R	ERRORS ON PSAMPLE FILE
	EQ	B0,B0,EXIT	RETURN
CALSMP	SA1	B5+3	GET FWA
	SA2	B5+4	GET LWA
	SX6	2315208	↓SMP↓
	LX6	42D	LEFT JUSTIFY ↓SMP↓
	MX0	42D	77777777777777000000
	BX1	-X0*X1	0..0FWA
	BX2	-X0*X2	0..0LWA
	LX1	18D	0..0FWA0..0
	IX6	X6+X1	SMP.FWA0..0
	BX6	X6+X2	SMP.FWA.LWA
	MX1	1	
	LX1	41D	RECALL BIT
	IX6	X6+X1	SMP.R.FWA.LWA
	RJ	=XSYS=	
	SX6	B0+1	TRUE
	SA6	B5+2	PSAMPLE := TRUE;
EXIT	SA1	B5+1	*****
	SB6	B5	*
	SB7	X1	* EXIT SEQUENCE
	LX1	42D	*
	SB5	X1	*
	JP	B7	*****
ERAD	BSS	4	
MESS1	DIS	,*PSAMPLE FILE NOT FOUND*	
MESS3	DIS	,*ERRORS ON PSAMPLE FILE*	
LEN	EQU	103B	*****
XXXXXX	FILEB	BUF,LEN,FET=14,EPR,PFN=PSAMPLE,USN=7ED	
BUF	BSS	LEN	*****
	END	SWLSMP	

SWLSMP INTERFACE FUNCTION - Figure 7.3 (Continued)

## GETTING STARTED

Assuming that you want to initiate your sample run from your terminal, you must create a SUBMIT file and give that file a name. The SUBMIT file should contain the following (the numbers on the left are not in the SUBMIT file they are for reference only).

1. XYZ,CM10000,T100. YOUR NAME
2. ACCOUNT,XYZ, PASSWORD.
3. CHARGE,CHARGENO,PROJECTID.
4. ATTACH,ISWL/UN=ALL.
5. GET,ISWLLIB/UN=ALL.
6. GET,UTL/UN=ALL.
7. GET,SWLSMP/UN=ZED.
8. GET,YOURPROG.
9. GET,YOURDATA.
10. RFL,70000.
11. COMPASS,I=SWLSMP,O=Z,B=LGO.
12. RFL,112000.
13. ISWL,I=YOURPROG,O=Z,B=LGO.
14. MAP,ON.
15. LGO,I=YOURDATA,O=Z.
16. PSAMP (,A).
17. REWIND,A.
18. COPYCR,A,Z.
19. REWIND,OUTPUT.
20. COPYCR,OUTPUT,Z.
21. CTIME.
22. DAYFILE,Z.
23. UTL,1000,INP=Z,OUTP=LIST.
24. DISPOSE,LIST=PR/EI=XYZ.
25. DAYFILE,LOOKSEE.
26. REPLACE,LOOKSEE.
27. EXIT.
28. DAYFILE,LOOKSEE.
29. REPLACE,LOOKSEE.

### SUBMIT FILE FOR SAMPLING - FIGURE 7.4

In the example, SUBMIT file in figure 7.4 the only things you need to change are those that are underlined. These include your user number (lines 1,2, and 24), your password (line 2), your charge information (line 3), the name of the file containing your program (lines 8 and 13), and the name of the file containing your data (lines 9 and 15). All the other lines may be used exactly as shown.

Lines 25 thru 29 produce a copy of the SUBMIT dayfile at your terminal. With this file (LOOKSEE) available you can examine the dayfile to be sure that all went well. The output listings will be available when you log-in to an Export/Import terminal (20OUT). If you want the output to be somewhere else, simply change line 24 accordingly.

## THE LOAD MAP

The load map will enable you to get a picture of where things are in the field length (memory) when your program is running.

Figure 7.5 is the load map from our sample program (to be discussed later).

Figure 7.6 shows how we can interpret the load map to gain a picture of where each module is loaded. This is necessary in order to correctly analyse the sample data.

LOAD MAP.

FL REQUIRED TO LOAD 13400  
FL REQUIRED TO RUN 6400  
INITIAL TRANSFER TO SWLSY.1 - 4055

BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE
SWLSMP	101	171	LGO
/GETPUT/	272	200	
LOOPS	472	164	LGO
SWLSYS	656	3350	ISWLLIB
CPUPFM	4226	10	SYSLIB
CPUSYS	4236	32	SYSLIB
//	4270	2002	

ENTRY POINTS.

ENTRY	ADDRESS	REFERENCES
SWLSMP		
SWLSMP	101	LOOPS 564
LOOPS		
MAIN	555	SWLSYS 4142
SWLSYS		
SWLSY.1	4055	
RPVTAB	4151	
CPUPFM		
PFM=	4227	SWLSMP 111
CPUSYS		
SYS=	4241	SWLSMP 133 CPUPFM 4226
RCL=	4253	SWLSMP 121
WNB=	4260	CPUPFM 4231
MSG=	4265	SWLSMP 116 125

TYPICAL LOAD MAP - Figure 7.5

0	KRONOS SYSTEM AREA
100	
101	SWLSMP INTERFACE FUNCTION
271	
272	/GET PUT/ COMMON I/O AREA
471	
472	LOOPS (THIS WILL BE YOUR PROGRAM)
655	
656	SWLSYS SWL SYSTEM ROUTINES
4225	
4226	CPUPFM CPU PERMANENT FILE MANAGER
4235	
4236	CPUSYS CPU SYSTEM MANAGER
4267	
4270	BLANK COMMON

MEMORY LAYOUT DERIVED FROM LOADER MAP - Figure 7.6

From the load map (fig. 7.5) or from the memory layout (fig. 7.6) we can see that our program "LOOPS" begins at location 472<sub>8</sub>. This number (472<sub>8</sub>) is called the (LOADER) program offset. It represents the offset from the start of the field length to the start of our program. This information is essential and only available from the loader map.

#### THE PROGRAM MEASURED

This will be your program, of course, but for this illustration we will supply a program that contains four SWL methods of accomplishing a repetitive loop with control variable. The four methods use: FOR, REPEAT, WHILE; and LOOP statements respectively. We will attempt to determine which method is the fastest and how much faster it is.

Figure 7.7 is a listing of the source program.

```

000001 VMODULE LOOPS:
000063 V
000063 VPROC [XREF] SWLSMP (VAL M,N:INTEGER) BOOLEAN:
000063 V
000063 VPROC [XDCL] MAIN:
000063 V
000063 VLABEL L1,L2,L3,L4:
000063 VCONST LOOPCOUNT = 100000:
000063 VVAR I : INTEGER:
000063 V
000063 VIF SWLSMP (0+472(8)+162(8)+472(8))
000072 V THEN WRITE(↑ SWLSMP OUTPUT FOLLOWS↑,EOL):
000112 V ELSE WRITE(↑ SWLSMP NOT AVAILABLE ↑,EOL):
000131 VTFEND:
000131 V
000131 VL1:FOR I := 1 TO LOOPCOUNT DO
000135 V   ≠FOR LOOP≠
000135 V   FOREND:
000140 V
000140 VL2:I:=0:
000141 V   REPEAT
000141 V     I:=I+1:
000143 V   ≠REPEAT LOOP≠
000143 V   UNTIL I >= LOOPCOUNT:
000145 V
000145 VL3:I:=0:
000146 V   WHILE I <= LOOPCOUNT DO
000150 V     I:=I+1:
000152 V   ≠WHILE LOOP≠
000152 V   WHILEEND:
000153 V
000153 VL4:I:=0:
000154 V   LOOP
000154 V     I:=I+1:
000156 V   ≠LOOP LOOP≠
000156 V   EXIT WHEN I >= LOOPCOUNT:
000161 V   LOOPEND:
000162 V
000162 VPROCEND MAIN:
000164 VMODULEEND LOOPS:

```

SOURCE PROGRAM TO BE SAMPLED - Figure 7.7

In figure 7.7 we have a sample program. The actual numbers on the left of the listing represent the offset (in words, OCTAL) of each statement from the beginning of the module. Each of the repetitive statements is labeled (L1,L2,L3 and L4). This insures that the statement will begin on a new word. We can see that the FOR statement (L1) begins at location 131<sub>8</sub> and continues until 137<sub>8</sub>. The repeat statement (L2) begins at 140<sub>8</sub>.

From the Loader Map we can see that the loader has loaded this module (LOOPS) beginning at 472<sub>8</sub>. Hence the actual location of the FOR statement is 131<sub>8</sub> to 137<sub>8</sub> plus 472<sub>8</sub> or 623<sub>8</sub> to 631<sub>8</sub>.

Figure 7.8 illustrates the computations required to find the actual locations of each of the repetitive statements.

1. BELOW FOR STATEMENT:	0 <sub>8</sub>	TO	130 <sub>8</sub>
+ LOADER OFFSET	472 <sub>8</sub>		472 <sub>8</sub>
<u>ACTUAL ADDRESSES</u>	472 <sub>8</sub>	TO	622 <sub>8</sub>
2. FOR STATEMENT:	131 <sub>8</sub>	TO	137 <sub>8</sub>
+ LOADER OFFSET	472 <sub>8</sub>		472 <sub>8</sub>
<u>ACTUAL ADDRESSES</u>	623 <sub>8</sub>	TO	631 <sub>8</sub>
3. REPEAT STATEMENT:	140 <sub>8</sub>	TO	144 <sub>8</sub>
+ LOADER OFFSET	472 <sub>8</sub>		472 <sub>8</sub>
<u>ACTUAL ADDRESSES</u>	632 <sub>8</sub>	TO	636 <sub>8</sub>
4. WHILE STATEMENT:	145 <sub>8</sub>	TO	152 <sub>8</sub>
+ LOADER OFFSET	472 <sub>8</sub>		472 <sub>8</sub>
<u>ACTUAL ADDRESSES</u>	637 <sub>8</sub>	TO	644 <sub>8</sub>
5. LOOP STATEMENT:	153 <sub>8</sub>	TO	162 <sub>8</sub>
+ LOADER OFFSET	472 <sub>8</sub>		472 <sub>8</sub>
<u>ACTUAL ADDRESSES</u>	645 <sub>8</sub>	TO	654 <sub>8</sub>
6. ABOVE LOOP STATEMENT :	655 <sub>8</sub>	TO	END

ACTUAL ADDRESS COMPUTATION - Figure 7.8

## ANALYZING THE RESULTS

The results of a sample run come in two parts. The first part is shown in figure 7.9 below.

```
P REGISTER SAMPLES FOR LGO,O=Z.  
75/03/10. 08.20.59.  
  
ELAPSED REAL TIME      10.645 SEC.  
ELAPSED CPU TIME       4.059 SEC.  
  
SAMPLE RANGE - FROM 000472 TO 000672  
  
SAMPLES JOB NOT ACTIVE      63695  
SAMPLES JOB IN RECALL       447  
SAMPLES BELOW RANGE         0  
SAMPLES IN RANGE            38235  
SAMPLES ABOVE RANGE         1635  
SAMPLES CPU AT SUR-CP       0  
  
TOTAL SAMPLES              104062
```

### SAMPLE OUTPUT - 1st PART - FIGURE 7.9

In the output shown in figure 7.9 the first line "P REGISTER SAMPLES FOR LGO,O=Z." indicates that all the information to follow is for the LGO statement only. This is the KRONOS control statement that caused our program (LOOPS) to begin execution. The samples are all done in the time between the invocation of SMP (By the statement IF SWLSMP (...)) until the "PROCEND MAIN;" statement in our program (LOOP).

The next line simply contains the date and time of this sample run.

The next two lines report the total real time and CPU time used while sampling took place.

The "SAMPLE RANGE" is given next as a check on the parameters (FWA & LWA) supplied in the call to function SWLSMP. Remember that the LWA will be rounded up to the next power of two interval (see fig. 7.2).

The remaining lines describe the results of the sampling (which occurs approximately once each  $100 \times 10^{-6}$  seconds).

The "SAMPLES JOB NOT ACTIVE" is the number of samples when the CPU was running some other job. During these samples real time is going by but there is no CPU time for our job.

"SAMPLES JOB IN RECALL" is the number of samples when our job was found to be in recall. This time usually represents I/O and system requests made by our job.

"SAMPLES BELOW RANGE" are the samples when the CPU was running our job but the P register was found below the range being sampled.

"SAMPLES IN RANGE" are the samples when the CPU was running our job and the P register was found to be within the range we have asked to sample.

"SAMPLES ABOVE RANGE" are the samples when the CPU was running our job, but the P register was found to be above the range being sampled.

"SAMPLES CPU AT SUB-CP" are the samples when the CPU was found at a sub-control point. These will always be zero unless we are sampling a KRONOS SUBSYSTEM (like TELEX, etc.).

"TOTAL SAMPLES" is just the sum of all the other reported samples.

Since the sampling rate is pretty close to one sample, each  $100 \times 10^{-6}$  second we can convert any sample count into time by multiplying the number of samples by  $10^{-4}$  ( $100 \times 10^{-6}$  sec =  $10^{-4}$  sec).

So, for example, the "TOTAL SAMPLES" is 104062. In terms of time, this is 10.4062 seconds. The "ELAPSED REAL TIME" was reported to be 10.645 seconds. The total samples should (in terms of time) equal the real time. Does 10.4062 equal 10.645? No! The error, however, is 0.2388 seconds or an error of  $((10.645 - 10.4062)/10.645) \times 100 = 2.24\%$  2.24 percent.

We could of course compute the sampling rate:

$$\frac{\text{REAL TIME}}{\text{TOTAL SAMPLES}} = \frac{10.645}{104062} = 102 \times 10^{-6} \text{ sec/sample}$$

But the result is so close to  $100 \times 10^{-6}$  seconds that we will generally use the nominal sampling rate of  $100 \times 10^{-6}$  sec/sample.

The sum of SAMPLES BELOW RANGE + SAMPLES IN RANGE + SAMPLES ABOVE RANGE will equal the ELAPSED CPU TIME (with a small percent error).

From the above information we can answer questions like:

1. What percent (or how much) real-time was spent outside our job (i.e. waiting on other jobs)?
2. What percent (or how much) real-time was spent waiting for our job to complete I/O & SYSTEM operations?
3. What percent (or how much) CPU time was spent in the range we are sampling?
4. Is our job I/O bound or CPU bound? What percent of the real time is I/O? What percent of the real time is CPU time?

There are, perhaps, many other questions we could ask. The above is just a sample of the kinds of information to be obtained from the first part of the output of a performance measurement run.

Figure 7.10 shows the second part of the sample output. This part is a detailed account of where the P register was found for samples ABOVE RANGE, IN RANGE, and BELOW RANGE.

The leftmost column is the P-register locations sampled. The next column is the number of samples (count) at each P-register location. The next column (PCT) is the percent of the CPU time that the count represents. Then we have a bunch of asterisks(\*). Each asterisk represents 1 percent. So the number of asterisks is a bar-chart (or histogram) representation of the percent (PCT) column. The plus signs (+), when connected, form an accumulation of CPU time. Notice the numbers 0 to 9, 0 along the top margin. These represent percent of CPU TIME accumulated. So after connecting the plus signs together we can make statements like: "By the time location 633<sub>8</sub> is reached 28 percent of the CPU time has been used". This kind of comment is only really meaningful for a program that progresses from top to bottom without a lot of jumps (or GOTO's) from one end of the program to the other.



In figure 7.10 the locations of each statement (FOR, REPEAT, WHILE, LOOP) are identified. By adding up the percentages we can determine how much (or what percent of) CPU time is spent on each statement. This information is summarized in figure 7.11.

1. BELOW	472 <sub>8</sub> TO 622 <sub>8</sub>	0.0%
2. FOR STATEMENT	623 <sub>8</sub> TO 631 <sub>8</sub>	21.7%
3. REPEAT STATEMENT	632 <sub>8</sub> TO 636 <sub>8</sub>	22.1%
4. WHILE STATEMENT	637 <sub>8</sub> TO 644 <sub>8</sub>	25.7%
5. LOOP STATEMENT	645 <sub>8</sub> TO 654 <sub>8</sub>	26.0%
6. ABOVE	655 <sub>8</sub> TO END	<u>4.2%</u>
TOTAL		99.7%

#### REPETITIVE STATEMENT RESULTS - FIGURE 7.11

As shown in figure 7.11 there isn't much difference between these repetitive statements, considering our 3% error factor. However, the fastest repetitive statement is the FOR (using 21.7% of the CPU TIME). The slowest repetitive statement is the LOOP statement (using 26.0% of the CPU TIME).

So there is a difference in repetitive statements - but not much.

#### IMPROVING PERFORMANCE

The techniques shown above can be used to determine how much CPU time is being spent in each Procedure in your program.

This can lead you quickly to examine the few procedures that are taking most of the CPU time. Experience has shown that most of the CPU time is generally spent in one or two of the many procedures in a software system.

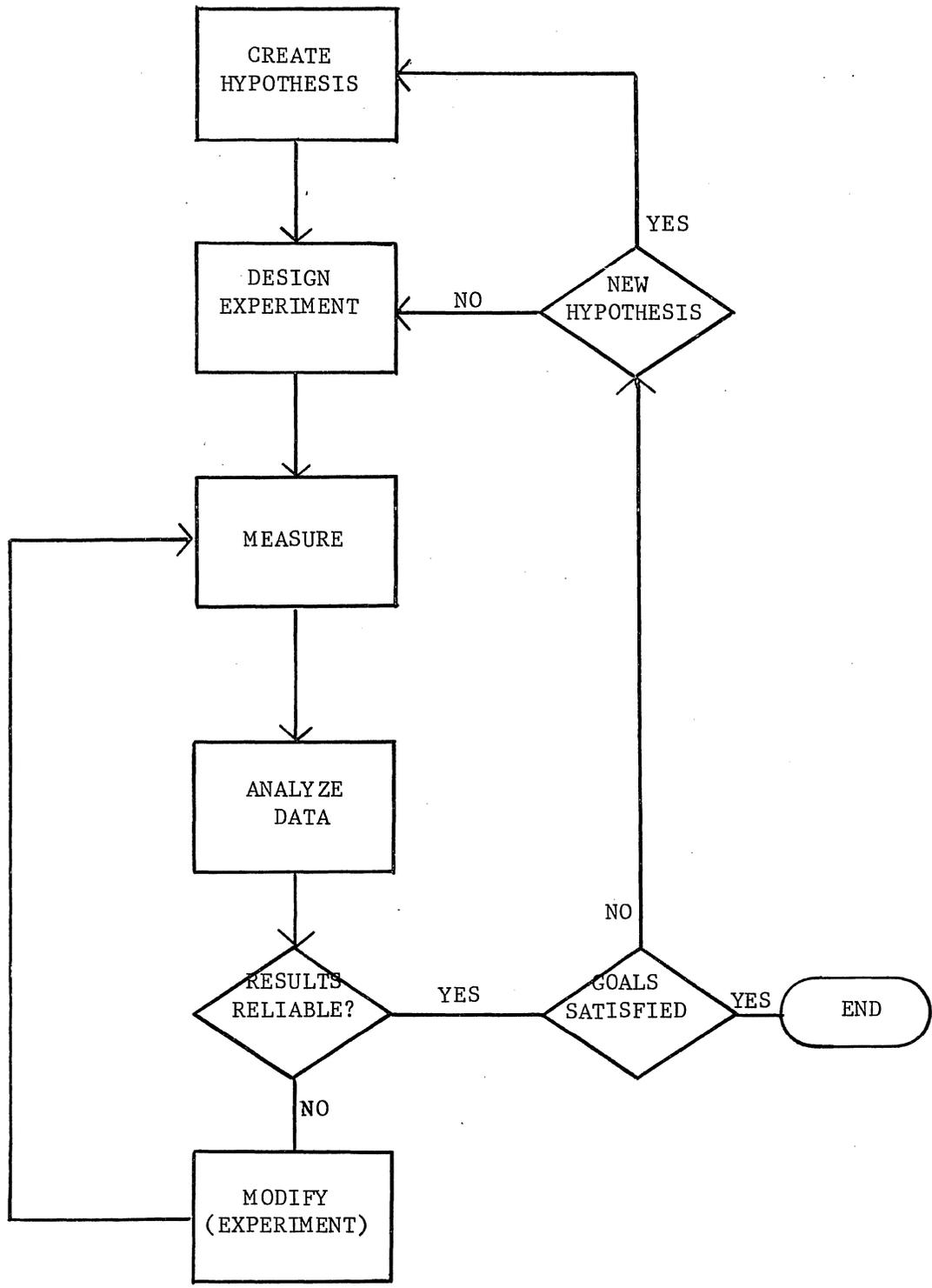
While each program is unique there are generally two ways of improving the performance of a program. One method is to fix up the executable code. This can be done using a different algorithm or eliminating redundant execution statements.

The other method for improving performance is to change the data structures and of course thereby change the access to the data structure. In many procedures significant amounts of CPU time are wasted thru inefficient accessing of data structures.

#### SUMMARY

With the tools and concepts presented here you can do a reasonably complete performance measurement study. You will be able to speed up your programs. The amount of speed up, of course, is difficult to say. The few software products analyzed thru performance measurement so far have been improved anywhere from 50 to 400 percent.

Sometimes many modifications are required to properly obtain a significant performance improvement. The flowchart in figure 7.12 illustrates how successive experiments are used to improve Software.



PERFORMANCE IMPROVEMENT CYCLE - FIGURE 7.12



APPENDIX A

SWL RESERVED WORD LIST



ALIGNED	NIL
ALLOCATE	NOT
AND	OF
ARRAY	OPEN
BEGIN	OR
BY	ORIF
CASE	PACKED
CASEND	POP
CAT	PROC
CLOSE	PROCEND
CODE	PUSH
CONST	QUEUE
COPROC	RECEIVED
CRAMMED	RECORD
CREATE	REF
CYCLE	REL
DEFINE	REP
DEQUEUE	REPDEP
DESTROY	REPEAT
DO	RESET
DOWNTO	RESUME
ELSE	RETURN
END	REWIND
ENQUEUE	SEGMENT
EXECUTE	SEQ
EXIT	SET
EXTERNAL	STACK
FILE	STATIC
FOR	STRING
FOREND	TAG
FREE	THEN
GOTO	TO
HEAP	TYPE
IF	UNION
IFEND	UNPACKED
IN	UNTIL
LABEL	VAL
LOOP	VAR
LOOPEND	WEOF
MACRO	WHEN
MACROEND	WHILE
MOD	WHILEND
MODE	XOCL
MODEND	XOR
MODULE	XREF
NEXT	



**APPENDIX B**

**ERROR LIST**



WL ERROR MESSAGES - 1 FEB 75

1: SCALAR TYPE EXPECTED.  
2: INTEGER TOO LARGE.  
3: ERROR IN CONSTANT.  
4: "=" EXPECTED.  
5: FIELD NAME DECLARED TWICE.  
6: BAD RANGE.  
7: TAG FIELD TYPE BAD.  
8: NAME DECLARED TWICE.  
9: ")" EXPECTED.  
0: ":" EXPECTED.  
1: IDENTIFIER EXPECTED.  
2: IDENTIFIER NOT DECLARED.  
3: INDEX MUST BE OF SCALAR TYPE.  
4: "OF" EXPECTED.  
5: TYPE IDENTIFIER EXPECTED.  
6: PROCEDURE DECLARED TWICE.  
7: "END" EXPECTED.  
8: ERROR IN TYPE DECLARATION.  
9: ERROR IN VARIABLE DECLARATION.  
0: ERROR IN INITIALIZATION VALUES.  
1: ERROR IN PROCEDURE DECLARATION.  
2: TOO MANY INITIALIZATION VALUES.  
3: PARAMETER LIST IGNORED.  
4: ERROR IN DECLARATION PART.  
5: LOWBOUND > HIGHBOUND.  
6: NOT A VARIABLE IDENTIFIER.  
7: DECREASING ADDRESSES IN VALUE PART.  
8: SYMBOLIC SUBRANGE TYPE NOT ALLOWED.  
9: PARAMETER MISSING IN FUNCTION DECLARATION.  
0: ILLEGAL COMPONENT TYPE.  
1: UNDECLARED IDENTIFIER.  
2: VARIABLE OR FIELD IDENTIFIER EXPECTED.  
3: EXPRESSION TOO COMPLICATED.  
4: TYPE OF VARIABLE SHOULD BE ARRAY.  
5: TYPE OF EXPRESSION MUST BE SCALAR.  
6: CONFLICT OF INDEX TYPE WITH DECLARATION.  
7: ")" EXPECTED.  
8: TYPE OF VARIABLE SHOULD BE RECORD.  
9: NO SUCH FIELD IN THIS RECORD.  
0: TYPE OF VARIABLE SHOULD BE POINTER OR FILE.  
1: FIELD NAME EXPECTED.  
2: ILLEGAL SYMBOL IN EXPRESSION.  
3: UNDEFINED LABEL.  
4: ILLEGAL TYPE OF PARAMETER IN STANDARD FUNCTION OR PROCEDURE.  
5: TYPE IDENTIFIER IN STATEMENT PART.  
6: PROCEDURE USED AS FUNCTION.  
7: INTEGER EXPRESSION EXPECTED.  
8: ")" EXPECTED.  
9: IDENTIFIER EXPECTED.  
0: ILLEGAL TYPE OF OPERAND.  
1: "OR" AND "XOR" CANNOT BE USED AS MONADIC OPERATORS.  
2: "!=" EXPECTED.  
3: ASSIGNMENT NOT ALLOWED.  
4: ILLEGAL SYMBOL IN STATEMENT.  
5: TYPE OR CONSTANT IDENTIFIER.  
6: "THEN" EXPECTED.  
7: TYPE OF EXPRESSION IS NOT BOOLEAN.  
8: ";" EXPECTED.  
9: "DO" EXPECTED.  
0: ILLEGAL PARAMETER SUBSTITUTION.  
1: LABEL EXPECTED.

62: ILLEGAL TYPE OF EXPRESSION.  
 63: CONSTANT EXPECTED.  
 64: ":" EXPECTED.  
 65: "OF" EXPECTED.  
 66: TAG FIELD MISSING FOR THIS VARIANT.  
 67: "UNTIL" EXPECTED.  
 68: "END" EXPECTED.  
 69: LOOP CONTROL VARIABLE MUST BE SIMPLE AND LOCAL OR GLOBAL.  
 70: "TO" OR "DOWNTO" EXPECTED.  
 71: TOO MANY CASES IN CASE STATEMENT.  
 72: NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION.  
 73: MIXED TYPES.  
 74: TOO MANY LABELS IN THIS PROCEDURE.  
 75: TOO MANY LONG CONSTANTS OR YET UNDEFINED LABELS IN THIS PROCEDURE.  
 76: DEPTH OF PROCEDURE NESTING TOO LARGE.  
 77: LABEL DEFINED MORE THAN ONCE.  
 78: TOO MANY EXIT LABELS.  
 79: "(" EXPECTED.  
 80: ", " EXPECTED.  
 81: ASSIGNMENT TO FORMAL FUNCTION IDENTIFIER ILLEGAL.  
 82: TOO MANY NESTED WITH-STATEMENTS.  
 83: STANDARD IN-LINE PROCEDURE OR FUNCTION USED AS ACTUAL PARAMETER.  
 84: TOO MANY LONG CONSTANTS IN THIS PROCEDURE.  
 85: ASSIGNMENT TO FUNCTION IDENTIFIER MUST OCCUR IN FUNCTION ITSELF.  
 86: ACTUAL PARAMETER MUST BE A VARIABLE.  
 87: VARIABLE MUST BE ALIGNED.  
 88: OPERATORS "<" AND ">" ARE NOT DEFINED FOR POWERSETS.  
 89: REDUNDANT OPERATION ON POWERSETS.  
 90: PROCEDURE TOO LONG.  
 91: TOO MANY EXIT LABELS OR FORWARD PROCEDURES.  
 92: TOO MANY POINTER TYPES OR FILE VARIABLES.  
 93: BAD FUNCTION TYPE.  
 94: ONLY "=" AND "#" ALLOWED HERE.  
 95: BAD FILE DECLARATION.  
 96: TYPE DECLARED TWICE.  
 97: "END." ENCOUNTERED.  
 98: "I" EXPECTED.  
 99: INDEX OUT OF RANGE.  
 100: LABEL TOO LARGE.  
 101: VALUE IS OUT OF RANGE.  
 102: DIVISION BY ZERO.  
 103: PARAMETER PROCEDURE HAS MORE THAN 17 PARAMETERS.  
 104: TEN OR MORE ERRORS ON THIS LINE.  
 105: ";" NOT ALLOWED IN COMMENTS, COMMENT TERMINATED.  
 106: IDENTIFIER TOO LONG.  
 107: "MODULE" EXPECTED.  
 108: "MODEND" EXPECTED.  
 109: INVALID ATTRIBUTE.  
 110: "IFEND" EXPECTED.  
 111: TOO MANY ALTERNATIVES IN IF STATEMENT.  
 112: "LOOPEND" EXPECTED.  
 113: "WHILEND" EXPECTED.  
 114: "FOREND" EXPECTED.  
 115: END LABEL DOES NOT MATCH.  
 116: NESTING OF STATEMENTS AND PROCEDURES TOO DEEP.  
 117: ILLEGAL REFERENCE TO LABEL OR PROCEDURE NAME.  
 118: TOO MANY UNRESOLVED REFERENCES.  
 119: "WHEN" EXPECTED.  
 120: MORE THAN ONE LABEL ON STATEMENT.  
 121: NO ENCLOSING PROCEDURE.  
 122: CYCLE REFERENCE NOT TO REPETITION STATEMENT.  
 123: "VAL" OR "REF" EXPECTED.  
 124: LOCAL CLASS MAY NOT BE REFERENCED BY EXTERNAL POINTER.  
 125: VARTARIF MUST BE STATIC OR LOCAL.

126: \*PROCEND\* EXPECTED.  
 127: SYSTEM ERROR (INVALID RELOCATION).  
 128: FEATURE NOT IMPLEMENTED.  
 129: \*RECELD\* EXPECTED.  
 130: STORAGE OR SCOPE ATTRIBUTE EXPECTED.  
 131: \*RECELD\* EXPECTED.  
 132: ILLEGAL BUILT-IN FUNCTION NAME.  
 133: ILLEGAL CONVERSION FUNCTION OR VALUE CONSTRUCTOR TYPE.  
 134: NOT SUPPORTED YET.  
 135: '(' OR 'I' EXPECTED.  
 136: ELEMENT VALUE OUT OF RANGE.  
 137: \*MODE\* NOT SUPPORTED.  
 138: FILE MAY NOT BE INITIALIZED.  
 139: POINTER MAY ONLY BE INITIALIZED TO NIL.  
 140: COMPILER ERROR (CTPTR = NIL IN CHECKTYPEBOUNDS).  
 141: SELECTION VALUE MUST BE ORDINAL, INTEGER, OR CHAR CONSTANT.  
 142: MULTIPLE SPECIFICATION OF SELECTION VALUES.  
 143: \*CASEND\* EXPECTED.  
 144: NUMBER OF INITIALIZATION VALUES EXCEEDS ISWL LIMITATION.  
 145: NUMBER OF REPLICATIONS LESS THAN 1.  
 146: ', ' OR ';' EXPECTED.  
 147: STRING LENGTH ERROR.  
 148: ILLEGAL OPERAND FOR CONCATENATION.  
 149: ILLEGAL CONVERSION FUNCTION IN CONSTANT.  
 150: POINTER VARIABLE EXPECTED.  
 151: STRING OR CHARACTER EXPRESSION EXPECTED.  
 152: CHAR EXPRESSION EXPECTED.  
 153: SUBSTRINGS OR STRING CONVERSIONS NOT ALLOWED AS PARAMETERS.  
 154: IDENTIFIERS IN PRONGS LIST NOT DECLARED.  
 155: ALL OR NONE OF IDENTIFIER LIST MUST BE PRONGS OF OUTERMOST MODULE.  
 156: MODULE NESTING ERROR.  
 157: ONLY VARS AND PROCS ALLOWED AS PRONGS OF OUTERMOST MODULE.  
 158: TOO MANY BLOCKS OR MODULES IN A COMPILATION UNIT; DEBUG TURNED OFF.  
 159: SIZE OF SEQUENCE TYPE IS ZERO WORDS.  
 160: SEQUENCE VARIABLE EXPECTED.  
 161: \*REP\* EXPECTED.  
 162: \*IN\* EXPECTED.  
 163: NOT NESTED IN A REPDEP PROC.  
 164: DECLARATION PART EXPECTED.  
 165: DEBUG STATEMENT-NUMBER TABLE OVERFLOW; DEBUG TURNED OFF.  
 166: DEBUG SYMBOL TABLE OVERFLOW; DEBUG TURNED OFF.  
 167: STRING TYPE VARIABLE EXPECTED.  
 168: DECIMAL SPECIFICATION IGNORED.  
 169: INITIALIZATION VALUE AND ELEMENT NOT OF EQUIVALENT TYPE.  
 170: VALUE MUST BE QUOTED STRING.  
 171: TOO MANY VALUES FOR SUB-ARRAY.  
 172: 'I' OR \*REP I OF\* EXPECTED.  
 173: INSUFFICIENT VALUES FOR SUB-ARRAY.  
 174: \*J\* OR ', ' EXPECTED.  
 175: INSUFFICIENT VALUES FOR RECORD.  
 176: TOO MANY VALUES FOR RECORD.  
 177: TAG FIELD MAY NOT BE INITIALIZED BY '\*'.  
 178: MUST BE A SET TYPE.  
 179: THE OPERAND IS NOT A SET OR BOOLEAN.  
 180: ILLEGAL CONSTANT FACTOR.  
 181: \*CAT\* OPERATOR NOT PERMITTED HERE.  
 182: \*PACKED\* ATTRIBUTE CAN ONLY BE USED FOR ARRAYS AND RECORDS.  
 183: CANNOT CHANGE PACKING OF PREVIOUSLY DEFINED TYPE.  
 184: DATA TYPE OF EXPRESSION TOO COMPLICATED.  
 185: POINTER TO PROC EXPECTED.  
 186: PROCEDURE HAS VALUE PARAMETERS.  
 187: FORWARD PROC ALREADY HAS ATTRIBUTES.  
 188: STATEMENT NOT NESTED IN PROCEDURE.  
 189: ADAPTABLE TYPE NOT ALLOWED HERE

- 190: MORE THAN 1 ADAPTABLE FIELD IN RECORD OR ADAPTABLE FIELD IN CASES.
- 191: POINTER TO TYPEID EXPECTED.
- 192: UNION VARIABLE EXPECTED.
- 193: NOT A MEMBER OF THIS UNION.
- 194: POINTER VARIABLE EXPECTED.
- 195: UNIONS ARE NOT EQUIVALENT.
- 196: UNION OR POINTER VARIABLE EXPECTED.
- 197: POINTER MAY NOT BE PASSED BY REF TO UNION PARAMETER.
- 198: POINTER VARIABLE EXPECTED ON LEFT OF ':=:'.
- 199: ARRAY VARIABLE EXPECTED.
- 200: WRONG DIMENSION SPECIFICATION.
- 201: '...' EXPECTED.
- 202: NON-COMPARABLE STRUCTURES.
- 203:
- 204: WARNING : FIELD ASSUMED TO BE ALIGNED.



APPENDIX C  
LANGUAGE SUMMARY



RESERVED WORDUSAGE

ALIGNED	
ALLOCATE	ALLOCATE P;
AND	REPEAT S UNTIL A AND B;
ARRAY	VAR X: ARRAY [1..10] OF INTEGER;
BEGIN	
BY	
CASE	CASE TAG OF =1= S; =2= S; CASEND;
CASEND	CASE TAG OF =1= S; =2= S; CASEND;
CAT	VAR S: [STATIC] STRING(6) OF CHAR := 'ABC' CAT 'DEF';
CLOSE	CLOSE (F1);
CODE	
CONST	CONST LIMIT = 100;
COPROC	
CRAMMED	
CREATE	
CYCLE	REPEAT S1,CYCLE WHEN I<10; S2; UNTIL I=100;
DEFINE	
DEQUEUE	
DESTROY	
DO	FOR I:= 1 TO 10 DO S1; FOREND;
DOWNTO	FOR I:= 100 DOWNTO 1 DO S1; FOREND;
ELSE	IF I<100 THEN S1; ELSE S2; IFEND;
END	
ENQUEUE	
EXECUTE	
EXIT	LOOP S1; EXIT WHEN I=10; S2; LOOPEND;
EXTERNAL	
FILE	VAR F1 [OUT] : FILE OF CHAR;
FOR	FOR I:= 1 TO 10 DO S1; FOREND;
FOREND	FOR I:= 10 DOWNTO 1 DO S1; FOREND
FREE	FREE P;
GOTO	GOTO L1;
HEAP	
IF	IF I<10 THEN S1; IFEND;
IFEND	IF I<=100 THEN S2; IFEND;
IN	B:= 5 IN SETA;
LABEL	LABEL L1,L2,L3,L4;
LOOP	LOOP S1; LOOPEND;
LOOPEND	LOOP S1; LOOPEND;
MACRO	
MACROEND	
MOD	
MODE	
MODEND	MODULE TEST;S1;S2; MODEND TEST;
MODULE	MODULE TEST;S1;S2; MODEND TEST;
NEXT	NEXT P IN SEQVAR;

## SWL LANGUAGE SUMMARY

Speed = 1 { FOR  
REPEAT  
WHILE  
LOOP

RESERVED WORDUSAGE

NIL	P:=NIL
NOT	IF NOT #EOF (INPUT) THEN S1; IFEND;
OF	VAR S:STRING (10) OF CHAR;
OPEN	OPEN (F1);
OR	B:= A OR B;
ORIF	IF B THEN S1; ORIF A THEN S2; IFEND;
PACKED	VAR X: PACKED ARRAY [1..10] OF 0..8;
POP	
PROC	PROC NAME; S1,S2; PROCEND;
PROCEND	PROC NAME; S1,S2; PROCEND;
PUSH	
QUEUE	
RECEND	TYPE R= RECORD F1,F2: INTEGER RECEND;
RECORD	TYPE R= RECORD F1,F2: INTEGER RECEND;
REF	PROC NAME (REF I:INTEGER); S1;S2; PROCEND;
REL	
REP	VAR X: [STATIC] ARRAY [1..10] OF INTEGER:= [REP 10 OF 0];
REPDEP	PROC [REPDEP] NAME; S1, S2; PROCEND;
REPEAT	REPEAT S1 UNTIL I<6;
RESET	RESET P;
RESUME	
RETURN	RETURN WHEN I<10;
REWIND	REWIND (INPUT);
SEGMENT	
SEQ	VAR Q:SEQ (REP 10 OF R);
SET	
STACK	
STATIC	VAR X: [STATIC] ARRAY [1..10] OF INTEGER:= [REP 10 OF 0];
STRING	VAR S: STRING (10) OF CHAR;
TAG	
THEN	IF I<10 THEN S1; IFEND;
TO	FOR I:= 1 TO 10 DO;
TYPE	TYPE ORD= (RED,BLUE,GREEN);
UNION	TYPE U= UNION (AReal, AINTEGER);
UNPACKED	
UNTIL	REPEAT S1 UNTIL I>=10;
VAL	PROC NAME (VAL I:INTEGER); S1,S2; PROCEND;
VAR	VAR I: INTEGER;
WEOF	WEOF (INPUT);
WHEN	EXIT WHEN I<10;
WHILE	WHILE I<10 DO S1;S2; WHILEND;
WHILEND	WHILE B DO S1; S2; WHILEND;
XDCL	PROC [XDCL] MAIN;
XOR	B:= A XOR B
XREF	PROC [XREF] SWLSMP;

APPENDIX D  
BIBLIOGRAPHY



## IMPLEMENTATION LANGUAGES

Software Writer's Language (SWL) Specifications  
CDC, 1975

MALUS Language Specifications  
CDC, 1972

SYMPL made simple:  
A CYBER SYMPT 1.1 User's Guide  
CDC, 1974

A. D. FALKOFF  
Criteria for a system design language  
in Software Engineering techniques.  
(J. N. BUXTON, B. RANDELL (eds.) )  
Report on a Conference sponsored by the NATOSCIENCE  
COMMITTEE, ROME, 1969

N. WIRTH  
PL360, a Programming Language for the  
360 Computers  
Journal of ACM 15,1 (Jan'68), pp 37-74

W. R. WULF, et al.  
BLISS: or Language for Systems Programming  
Comm ACM 14,12 (Dec'71), 780-790

G. SEEGMULLER  
Systems Programming as an Emerging Discipline  
Proceedings IFIP Congress 74, pp 417-426  
North-Holland Publ. Comp., Amsterdam, 1974

PASCAL

W. F. BURGER  
PASCAL Manual

Dept. of Comp. Science, Univ. of Texas at Austin, March '73

C. A. R. HOARE, N. WIRTH

An Axiomatic Definition of the Programming  
Language PASCAL

Acta Informatica 2,4 (1973), pp 335-355

or

Berichte der Fachgruppe Computer-Wissenschaften  
Nr. 6 (Nov. 1972)

K. JENSEN, N. WIRTH

A User Manual for Pascal

Eidgenossische Technische Hochschule (ETH),  
ZURICH, Switzerland

G. H. RICHMOND (ed.)

PASCAL Newsletters, January '74, Number 1

SIGPLAN Notices 9,3 (March '74), 21-28

N. WIRTH

The Programming Language PASCAL

Acta Informatica 1,1 (1971), pp 35-63

or

Berichte der Fachgruppe Computer-Wissenschaften 1 (Nov.'70)

N. WIRTH

The design of a Pascal Compiler

Software-Practices and Experience Vol 1, 309-333 (1971)

N. WIRTH

The Programming Language PASCAL (revised report)

Eidgenossische Technische Hochschule (ETH) Zurich,  
Switzerland, July 1973

PROGRAMMING MANAGEMENT

F. T. BAKER

Chief Programmer Team Management of Production Programming  
IBM SYSTEM Journal 11,1, pp 56-73, 1972

F. T. BAKER

Systems Quality Through Structured Programming  
AFIPS Fall Joint Computer Conference Proceedings  
Vol 41, Part I, 1972, pp 337-343

F. T. BAKER, H. MILLS

Chief Programmer Teams  
Datamation 19, Dec. 1973, pp 58-61

F. L. BAUER (ed.)

Advanced Course in Software Engineering  
Feature notes in Economics and Mathematical  
Systems, 81, Springer Verlag, N.Y., 1973

J. DIRNBERGER, R. ROTHSTEIN

Fostering a Structured Programming Environment  
Principles of Software Development, Conf. Proc. CDC,  
1974, pp 132 - 175

H. MILLS

Chief Programmer Teams: Principles & Procedures  
IBM, Federal Systems Division, Ref. # FSC 71-5108,  
Gaithersburg, Md., June 1971

G. M. WEINBERG

The Psychology of Computer Programming  
Van Nostrand Reinhold, N.Y., 1971

G. M. WEINBERG

The Psychology of Improved Programming Performance  
Datamation 17,11 (Nov.'72), pp 81-85

## STRUCTURED PROGRAMMING

C. BOHM, G. JACOPINI

Flow diagrams, turning machines and languages  
with only two formation rules  
Comm. ACM 9,3, pp. 366-371

O.-J. DAHL, E. W. DIJKSTRA, C.A.R. HOARE

Structured Programming  
Academic Press, N.Y., 1972

E. W. DIJKSTRA

Structured Programming  
in Software Engineering Techniques,  
Report on a Conference sponsored by the  
NATO SCIENCE COMMITTEE, ROME, 1969, pp. 88-93  
(J.N. BUXTON, B. RANDELL (eds.) )

J. R. DONALDSON

The revolution in programming:  
Some advanced techniques  
Control Data Corp., Publ. #:60417000, 1973

EDP - ANALYSER

The Advent of Structured Programming  
Canning Publ. Comp., Vol. 12, 6 (June '74)

D. GRIES

On Structured Programming - A Reply to Smoliar  
Comm. ACM 17,11 (Nov.'74)

B. W. KERNIGHAN, P. J. PLAUGER

The Elements of Programming Style  
McGraw-Hill, 1974

D. E. KNUTH

Structured Programming with GOTO Statements  
ACM Computing Surveys 6,4 (Dec.'74), pp.261-302

B. H. LISKOV

A Design Methodology for Reliable Software Systems  
AFIPS Conference Proceedings 41 (Fall'72), pp. 191-199

D. L. PARNAS

On The Criteria to be Used in Decomposing Systems into Moduls  
Comm. ACM 15,12 (Dec.'72), pp. 1053-1058

N. WIRTH

Systematic Programming  
Prentice-Hall, 1973

**APPENDIX E**

**ANNOTATED TERMINAL SESSION**



75/03/13. 10.43.55.

MIDWEST CLUSTER CENTER/SYSTEM F

KRONOS 2.1.0-06

USER NUMBER: ED,R325T1

TERMINAL: 101,713

RECOVER/ CHARGE:CHARGE,IB726MX,\*SWL\*

READY.

1. LOGIN PROCEDURES.

HALF

READY.

2. THIS TERMINAL RUNS IN HALF-DUPLEX. THIS COMMAND TELLS KRONOS.

NEW,SUM

READY.

3. SPECIFY NEW PRIMARY FILE NAMED SUM.

TEXT

ENTER TEXT MODE.

MODULE SUM

PROC [XDCL] MAIN;

VAR I:[STATIC] INTEGER := 0

4. USING TEXT MODE ENTER ENTIRE PROGRAM. TERMINATE TEXT MODE WITH BREAK KEY.

"SKIP EMPTY LINES"

WHILE #EOF(INPUT) DO

WHILEND;

5. PACK COMMAND REQUIRED AFTER TEXT MODE INPUT.

"READ & SUM INPUT LINE"

READ(N)

WHILE NOT #EOF(INPUT) DO

I=I+N;

READ(N)

WHILEND;

6. SAVE SOURCE PROGRAM AS INDIRECT FILE NAMED SUM.

"WRITE RESULT"

WRITE(' SUM = ',I,EOL);

MODEN D SUM;

EXIT TEXT MODE.

PACK

READY.

SAVE

READY.

```
BATCH
SRFL,20000.
/ATTACH,ISWL/UN=ALL
/GET,ISWLLIB/UN=ALL
/RFL,105000
RFL,105000.
/ISWL,I=SUM
```

7. OBTAIN COPY OF COMPILER AND RUN TIME ROUTINES.

```
1
OCDC CYBER SYSTEM                ISWL 2.0        LEVEL 1
75/03/13.                        10.50.45.      PAGE 1
```

```
000001 !MODULE SUM
000063 !PR0C [XDCL] MAIN;
****          ↑58
000063 !VAR I:[STATIC] INTEGER := 0
000064 !
000064 !"SKIP EMPTY LINES"
000064 !WHILE #EOF(INPUT) D0
****          ↑146
000070 !WHILEND;
000071 !
000071 !"READ & SUM INPUT LINE"
000071 !READ(N)
****          ↑31
000071 !WHILE NOT #EOF(INPUT) D0
****          ↑58
000072 !I=I+N;
****          ↑52
000072 !READ(N)
****          ↑31
000072 !WHILEND;
000073 !
000073 !"WRITE RESULT"
000073 !WRITE(' SUM = ',I,E0L);
000107 !
000107 !M0DEN D SUM;
****          ↑31      ↑52
000107 !
```

8. "ISWL,I=SUM" CAUSES COMPILER TO COMPILE SOURCE PROGRAM AND PRINT LISTING AT THE TERMINAL. ERROR LINES ARE SHOWN.

```
9C * STK FRAME ADDR. = 105263
```

9. COMPILER "BLOWS UP".

- EDIT

BEGIN TEXT EDITING.

? L

MØDULE SUM

? RS:/SUM/,/SUM;/

? S;1

? L

PRØC [XDCL] MAIN;

? S;1

? L

VAR I:[STATIC] INTEGER := 0

? RS:/0/,/0;/

? F:/READ/

"READ & SUM INPUT LINE"

? S;1

? L

READ(N)

? RS:)/,)/;/

? F:/MØD/

MØDEN D SUM;

? RS:/N D/,/ND/

? END

10. BEGIN TEXT EDITING THE  
PRIMARY FILE CORRECTING  
ERRORS.

11. WHEN EDITING COMPLETE  
REPLACE OLD PERMANENT  
FILE COPY WITH REVISED  
(EDITED) COPY.

- END TEXT EDITING.

\$EDIT,SUM.

-/REPLACE

-/

ANNOTATED TERMINAL SESSION

12	<pre> [/ISWL,I=SUM,0=Z * NUMBER OF SYNTAX ERRORS = 5 /EDIT,Z BEGIN TEXT EDITING. ? F:/****/ </pre>	12. RECOMPILE EDITED PRIMARY FILE (SUM) AND PLACE LISTING ON FILE 2.
13	<pre> ****          +31 ? S;-1 ? L;3 000071 !READ(N); ****          +31 000071 !WHILE NOT #EOF(INPUT) D0 ? END END TEXT EDITING. \$EDIT,Z. /EDIT BEGIN TEXT EDITING. ? F:/VAR/ </pre>	13. SINCE THERE ARE ERRORS, EDIT FILE 2, FIND ERROR.
14	<pre> VAR I:[STATIC] INTEGER := 0; ? RS:/0;/,/0,/ ? ADD ENTER TEXT. ? ' N:INTEGER;' READY. ? END END TEXT EDITING. \$EDIT,SUM. </pre>	14. TO CORRECT ERROR WE MUST EDIT THE PRIMARY FILE.
15	<pre> [/REPLACE / </pre>	15. REPLACE PERMANENT FILE WITH NEW PRIMARY FILE.

ANNOTATED TERMINAL SESSION

```

[RETURN,LG0,Z
[RETURN,LG0,Z.
7 [ /ISWL,I=SUM,0=Z
[ * NUMBER OF SYNTAX ERRORS = 3
[ /EDIT,Z
16. RETURN OBJECT FILE (LGO)
AND LISTING FILE (2). THIS IS
A QUICK WAY OF EMPTYING THESE FILES.
BEGIN TEXT EDITING.
? L;*
1
OCDL CYBER SYSTEM ISWL 2.0 LEVEL 1
75/03/13. 10.58.30. PAGE 1
000001 !MODULE SUM;
000063 !PR0C [XDCL] MAIN;
000063 !VAR I:[STATIC] INTEGER := 0, 17. RECOMPILE WITH LISTINGS ON
000064 ! N:INTEGER; FILE 2.
000064 !
000064 !"SKIP EMPTY LINES"
000064 !WHILE #E0F(INPUT) D0
000070 !WHILEND;
000071 !
18. EDIT FILE 2 AND OBTAIN A
000071 !"READ & SUM INPUT LINE" COMPUTE COMPILATION LISTING.
000071 !READ(N);
000073 !WHILE NOT #E0F(INPUT) D0
000074 !I=I+N;
**** +52
000074 !READ(N)
000076 !WHILEND;
000077 !
000077 !"WRITE RESULT"
000077 !WRITE(' SUM = ',I,E0L);
000113 !
000113 !M0DEND SUM;
**** +126,58
-END OF FILE-
[? END
END TEXT EDITING.
$EDIT,Z.
/

```

	<pre>EDIT BEGIN TEXT EDITING. ? F:/I=I/ I=I+N;</pre>	19. MAKE CORRECTIONS TO EDIT FILE.
19	<pre>? RS:/=/,/:=/ ? F+ *DEL* F:/MØDEND/ MØDEND SUM; ? S;-1 ? ADD</pre>	
	<pre>ENTER TEXT. ? 'PRØCEND MAIN; ' READY. ? END END TEXT EDITING. \$EDIT,SUM.</pre>	20. REPLACE PERMANENT FILE WITH CONTENTS OF PRIMARY FILE.
20	<pre>/REPLACE /</pre>	
21	<pre>RETURN,LØ *DEL* RETURN,LGØ,Z</pre>	21. CLEAR OUT FILES LGO & 2.
22	<pre>\$RETURN,LGØ,Z. /ISWL,I=SUM,Ø=Z - END COMPILATION</pre>	22. RECOMPILE PRIMARY FILE. NOTICE NO COMPILATION ERRORS.
23	<pre>/LGØ ? 3 5 7 9 SUM = 24 - END EXECUTION</pre>	23. EXECUTE PROGRAM. INPUT DATA AFTER?
24	<pre>/LGØ ?</pre>	24. EXECUTE AGAIN. THIS TIME TEST AN EMPTY INPUT LINE.
25	<pre>*INTERRUPTED* X *TERMINATED*</pre>	25. AFTER A FEW MINUTES WITH NO RESPONSE WE REMEMBER THAT THE "REWIND(INPUT)" STATEMENT WAS OMITTED. BREAK KEY IS USED TO TERMINATE EXECUTION.
26	<pre>/EDIT BEGIN TEXT EDITING. ? F:/WHILE/ WHILE #EØF(INPUT) DØ ? L;3 WHILE #EØF(INPUT) DØ WHILEND;</pre>	26. USE EDIT TO ADD MISSING LINE IN PRIMARY FILE.
	<pre>? ADD ENTER TEXT. ? 'REWIND(INPUT); ' READY. ? END END TEXT EDITING. \$EDIT,SUM.</pre>	27. REPLACE PERMANENT FILE. EMPTY LGO & 2.
27	<pre>/REPLACE /RETURN,LGØ,Z \$RETURN,LGØ,Z. /ISWL,I=SUM,Ø=Z - END COMPILATION</pre>	RECOMPILE.

ANNOTATED TERMINAL SESSION

```

LGØ
? 2 3 4 5
SUM =          14          28. TEST PROGRAM WITH INPUT DATA.
- END EXECUTION
/LGØ
?
?
? 33 44 55
SUM =          132
- END EXECUTION
/SAVE, LGØ=SUMLGØ      30. SINCE PROGRAM WORKS SAVE THE
/BYE                   BINARY OBJECT DECK AS A FILE
                       CALLED SUMLGO FOR FUTURE USE.
ZED      LOG OFF.   11.06.02.
ZED      SS        10.369 SEC.      31. LOGOFF.
TIØ      3532

```

ANNOTATED TERMINAL SESSION



**APPENDIX F**  
**COMPILATION LISTING**



COMPILATION  
DATE  
YY/MM/DD

TIME OF  
COMPILATION  
HH.MM.SS.

COMPILER & VERSION

COMPUTER SYSTEM

1  
OCDC CYBER SYSTEM  
75/03/12.

19.45.09.

ISWL 2.0

LEVEL 1  
PAGE 1

```
000001 !MODULE LØØPS;
000063 !
-----
000063 !PRØC [XREF] SWLSMP (VAL M,N:INTEGER) BØØLEAN;
000063 !
000063 !PRØC [XDCL] MAIN;
000063 !
000063 !LABEL L1,L2;
000063 !CØNST LØPCØUNT = 100000;
-----
000063 !VAR I : INTEGER;
000063 !
000063 !IF SWLSMP (0+472(8),162(8)+472(8))
000072 ! THEN WRITE(' SWLSMP ØUTPUT FØLLØWS',EØL);
000112 ! ELSE WRITE(' SWLSMP NØT AVAILABLE ',EØL);
000131 !IFEND;
-----
000131 !
000131 !L1:FØR I := 1 TØ LØPCØUNT DØ
**** ←-----↑31
000132 ! "FØR LØØP"
000132 ! FØREND;
000135 !
-----
000135 !L2:I:=0;
000136 ! REPEAT
000136 ! I:=I+1;
000140 ! "REPEAT LØØP"
000140 ! UNTIL I >= LØPCØUNT;
**** ↑31
-----
000141 !
000141 !PRØCEND MAIN;
000143 !MØDEND LØØPS;
* NUMBER ØF SYNTAX ERRØRS = 2
```

ERROR  
DESIGNATION

ERROR  
SUMMARY

LOCATION OF SOURCE STATEMENT  
FROM BEGINNING OF MODULE (IN OCTAL).



APPENDIX G  
CHARACTER CODES

The character codes illustrated are specifically for the CDC 713 Terminals. Since the 713 Terminal follows the standard ASCII character coding, the codes shown will probably be the same for any ASCII Terminal.



! "\$K+ \$M80

! MODULE CHARACTER\$SET\$713;

! " DISPLAY CODES FOR CDC 713 TERMINALS  
! " DRIVEN UNDER KRONOS 2.1 LEVEL 6

! CONST

! NAME	VALUE	DESCRIPTION	ASCII	OPR	CHR	GRP	! "
! NUL	= \$CHAR( 0),	"NULL OR IDLE	0000 0000	N	N		! "
! SOH	= \$CHAR( 1),	"START OF MESSAGE	0000 0001	N	N		! "
! STX	= \$CHAR( 2),	"START OF TEXT	0000 0010	N	N		! "
! ETX	= \$CHAR( 3),	"END OF TEXT	0000 0011	Y	Y		! "
! EOT	= \$CHAR( 4),	"END OF TRANSMISSION	0000 0100	N	N		! "
! ENQ	= \$CHAR( 5),	"ENQUIRY	0000 0101	N	N		! "
! ACK	= \$CHAR( 6),	"ACKNOWLEDGE	0000 0110	N	N		! "
! BELL	= \$CHAR( 7),	"BELL	0000 0111	N	N		! "
! BS	= \$CHAR( 8),	"CURSOR LEFT (BACKSPACE)	0000 1000	Y	N		! "
! HT	= \$CHAR( 9),	"HORIZONTAL TAB	0000 1001	N	N		! "
! LF	= \$CHAR( 10),	"LINE FEED (EOL)	0000 1010	Y	N		! "
! VT	= \$CHAR( 11),	"VERTICAL TAB	0000 1011	N	N		! "
! FF	= \$CHAR( 12),	"FORM FEED	0000 1100	N	N		! "
! CR	= \$CHAR( 13),	"CARRIAGE RETURN	0000 1101	Y	N		! "
! SO	= \$CHAR( 14),	"SHIFT OUT (BLACK ON WHITE)	0000 1110	Y	Y		! "
! SI	= \$CHAR( 15),	"SHIFT IN (WHITE ON BLACK)	0000 1111	Y	Y		! "
! DLE	= \$CHAR( 16),	"DEVICE CONTROL 1	0001 0000	N	N		! "
! DC1	= \$CHAR( 17),	"DEVICE CONTROL 2	0001 0001	N	N		! "
! DC2	= \$CHAR( 18),	"DEVICE CONTROL 3	0001 0010	N	N		! "
! DC3	= \$CHAR( 19),	"DEVICE CONTROL 4	0001 0011	N	N		! "
! DC4	= \$CHAR( 20),	"DEVICE CONTROL 4	0001 0100	N	N		! "
! SKIP	= \$CHAR( 21),	"CURSOR RIGHT (SKIP)	0001 0101	Y	N		! "
! LCLR	= \$CHAR( 22),	"LINE CLEAR	0001 0110	Y	N		! "
! ETB	= \$CHAR( 23),	"END TRANSMISSION BLOCK	0001 0111	N	N		! "
! CLR	= \$CHAR( 24),	"CLEAR SCREEN	0001 1000	Y	N		! "
! RSET	= \$CHAR( 25),	"RESET CURSOR (TO HOME)	0001 1001	Y	N		! "
! CUP	= \$CHAR( 26),	"CURSOR UP	0001 1010	Y	N		! "
! ESC	= \$CHAR( 27),	"ESCAPE	0001 1011	N	N		! "
! FS	= \$CHAR( 28),	"FIELD SEPARATOR	0001 1100	N	N		! "

!	GS	= \$CHAR( 29),	"GROUP SEPARATOR	0001	1101	N	N	"!
!	RS	= \$CHAR( 30),	"RECORD SEPARATOR	0001	1110	N	N	"!
!	US	= \$CHAR( 31),	"UNIT SEPARATOR	0001	1111	N	N	"!
!	SP	= \$CHAR( 32),	"SPACE	0010	0000	Y	N	"!
!	EX	= \$CHAR( 33),	"EXCLAMATION MARK	0010	0001	Y	Y	"!
!	DQ	= \$CHAR( 34),	"DOUBLE QUOTE	0010	0010	Y	Y	"!
!	PS	= \$CHAR( 35),	"NUMBER SIGN	0010	0011	Y	Y	"!
!	DS	= \$CHAR( 36),	"DOLLAR SIGN	0010	0100	Y	Y	"!
!	PER	= \$CHAR( 37),	"PERCENT SIGN	0010	0101	Y	Y	"!
!	AMP	= \$CHAR( 38),	"AMPERSAND	0010	0110	Y	Y	"!
!	RAP	= \$CHAR( 39),	"RIGHT APOSTROPHE	0010	0111	Y	Y	"!
!	LP	= \$CHAR( 40),	"LEFT PARENTHESIS	0010	1000	Y	Y	"!
!	RP	= \$CHAR( 41),	"RIGHT PARENTHESIS	0010	1001	Y	Y	"!
!	AS	= \$CHAR( 42),	"ASTERISK	0010	1010	Y	Y	"!
!	PL	= \$CHAR( 43),	"PLUS	0010	1011	Y	Y	"!
!	CM	= \$CHAR( 44),	"COMMA	0010	1100	Y	Y	"!
!	MI	= \$CHAR( 45),	"MINUS	0010	1101	Y	Y	"!
!	PERD	= \$CHAR( 46),	"PERIOD	0010	1110	Y	Y	"!
!	SOL	= \$CHAR( 47),	"SLASH OR SOLIDUS	0010	1111	Y	Y	"!
!	ZER	= \$CHAR( 48),	"ZERO	0011	0000	Y	Y	"!
!	ONE	= \$CHAR( 49),	"ONE	0011	0001	Y	Y	"!
!	TWO	= \$CHAR( 50),	"TWO	0011	0010	Y	Y	"!
!	THR	= \$CHAR( 51),	"THREE	0011	0011	Y	Y	"!
!	FOUR	= \$CHAR( 52),	"FOUR	0011	0100	Y	Y	"!
!	FIVE	= \$CHAR( 53),	"FIVE	0011	0101	Y	Y	"!
!	SIX	= \$CHAR( 54),	"SIX	0011	0110	Y	Y	"!
!	SEV	= \$CHAR( 55),	"SEVEN	0011	0111	Y	Y	"!
!	EGH	= \$CHAR( 56),	"EIGHT	0011	1000	Y	Y	"!
!	NIN	= \$CHAR( 57),	"NINE	0011	1001	Y	Y	"!
!	COL	= \$CHAR( 58),	"COLON	0011	1010	Y	Y	"!
!	SC	= \$CHAR( 59),	"SEMI-COLON	0011	1011	Y	Y	"!
!	LT	= \$CHAR( 60),	"LESS THAN	0011	1100	Y	Y	"!
!	EQ	= \$CHAR( 61),	"EQUAL	0011	1101	Y	Y	"!
!	GT	= \$CHAR( 62),	"GREATER THAN	0011	1110	Y	Y	"!
!	QM	= \$CHAR( 63),	"QUESTION MARK	0011	1111	Y	Y	"!
!	AT	= \$CHAR( 64),	"AT SIGN	0100	0000	Y	Y	"!
!	A	= \$CHAR( 65),	"UPPER CASE A	0100	0001	Y	Y	"!
!	B	= \$CHAR( 66),	"UPPER CASE B	0100	0010	Y	Y	"!
!	C	= \$CHAR( 67),	"UPPER CASE C	0100	0011	Y	Y	"!
!	D	= \$CHAR( 68),	"UPPER CASE D	0100	0100	Y	Y	"!
!	E	= \$CHAR( 69),	"UPPER CASE E	0100	0101	Y	Y	"!

!	F	= \$CHAR( 70),	"UPPER CASE F	0100	0110	Y	Y	"!
!	G	= \$CHAR( 71),	"UPPER CASE G	0100	0111	Y	Y	"!
!	H	= \$CHAR( 72),	"UPPER CASE H	0100	1000	Y	Y	"!
!	I	= \$CHAR( 73),	"UPPER CASE I	0100	1001	Y	Y	"!
!	J	= \$CHAR( 74),	"UPPER CASE J	0100	1010	Y	Y	"!
!	K	= \$CHAR( 75),	"UPPER CASE K	0100	1011	Y	Y	"!
!	L	= \$CHAR( 76),	"UPPER CASE L	0100	1100	Y	Y	"!
!	M	= \$CHAR( 77),	"UPPER CASE M	0100	1101	Y	Y	"!
!	N	= \$CHAR( 78),	"UPPER CASE N	0100	1110	Y	Y	"!
!	O	= \$CHAR( 79),	"UPPER CASE O	0100	1111	Y	Y	"!
!	P	= \$CHAR( 80),	"UPPER CASE P	0101	0000	Y	Y	"!
!	Q	= \$CHAR( 81),	"UPPER CASE Q	0101	0001	Y	Y	"!
!	R	= \$CHAR( 82),	"UPPER CASE R	0101	0010	Y	Y	"!
!	S	= \$CHAR( 83),	"UPPER CASE S	0101	0011	Y	Y	"!
!	T	= \$CHAR( 84),	"UPPER CASE T	0100	0100	Y	Y	"!
!	U	= \$CHAR( 85),	"UPPER CASE U	0100	0101	Y	Y	"!
!	V	= \$CHAR( 86),	"UPPER CASE V	0100	0110	Y	Y	"!
!	W	= \$CHAR( 87),	"UPPER CASE W	0100	0111	Y	Y	"!
!	X	= \$CHAR( 88),	"UPPER CASE X	0100	1000	Y	Y	"!
!	Y	= \$CHAR( 89),	"UPPER CASE Y	0100	1001	Y	Y	"!
!	Z	= \$CHAR( 90),	"UPPER CASE Z	0100	1010	Y	Y	"!
!	OB	= \$CHAR( 91),	"OPEN BRACKET	0100	1011	Y	Y	"!
!	BSH	= \$CHAR( 92),	"BACK SLASH	0100	1100	Y	Y	"!
!	CB	= \$CHAR( 93),	"CLOSE BRACKET	0100	1101	Y	Y	"!
!	CIR	= \$CHAR( 94),	"CIRCUMFLEX	0100	1110	Y	Y	"!
!	UND	= \$CHAR( 95),	"UNDERScore	0100	1111	Y	Y	"!
!	LSQ	= \$CHAR( 96),	"LEFT SINGLE QUOTE	0101	0000	Y	Y	"!
!	LCA	= \$CHAR( 97),	"LOWER CASE A	0101	0001	Y	Y	"!
!	LCB	= \$CHAR( 98),	"LOWER CASE B	0101	0010	Y	Y	"!
!	LCC	= \$CHAR( 99),	"LOWER CASE C	0101	0011	Y	Y	"!
!	LCD	= \$CHAR(100),	"LOWER CASE D	0101	0100	Y	Y	"!
!	LCE	= \$CHAR(101),	"LOWER CASE E	0101	0101	Y	Y	"!
!	LCF	= \$CHAR(102),	"LOWER CASE F	0101	0110	Y	Y	"!
!	LCG	= \$CHAR(103),	"LOWER CASE G	0101	0111	Y	Y	"!
!	LCH	= \$CHAR(104),	"LOWER CASE H	0101	1000	Y	Y	"!
!	LCI	= \$CHAR(105),	"LOWER CASE I	0101	1001	Y	Y	"!
!	LCJ	= \$CHAR(106),	"LOWER CASE J	0101	1010	Y	Y	"!
!	LCK	= \$CHAR(107),	"LOWER CASE K	0101	1011	Y	Y	"!
!	LCL	= \$CHAR(108),	"LOWER CASE L	0101	1100	Y	Y	"!
!	LCM	= \$CHAR(109),	"LOWER CASE M	0101	1101	Y	Y	"!
!	LCN	= \$CHAR(110),	"LOWER CASE N	0101	1110	Y	Y	"!

```

! LCO = $CHAR(111), "LOWER CASE O 0101 1111 Y Y ""!
! LCP = $CHAR(112), "LOWER CASE P 0110 0000 Y Y ""!
! LCQ = $CHAR(113), "LOWER CASE Q 0110 0001 Y Y ""!
! LCR = $CHAR(114), "LOWER CASE R 0110 0010 Y Y ""!
! LCS = $CHAR(115), "LOWER CASE S 0110 0011 Y Y ""!
! LCT = $CHAR(116), "LOWER CASE T 0110 0100 Y Y ""!
! LCU = $CHAR(117), "LOWER CASE U 0110 0101 Y Y ""!
! LCV = $CHAR(118), "LOWER CASE V 0110 0110 Y Y ""!
! LCW = $CHAR(119), "LOWER CASE W 0110 0111 Y Y ""!
! LCX = $CHAR(129), "LOWER CASE X 0110 1000 Y Y ""!
! LCY = $CHAR(130), "LOWER CASE Y 0110 1001 Y Y ""!
! LCZ = $CHAR(131), "LOWER CASE Z 0110 1010 Y Y ""!
! OBR = $CHAR(123), "OPEN BRACE 0110 1011 Y Y ""!
! SOR = $CHAR(124), "STYLIZED OR (VERTICAL BAR) 0110 1100 Y Y ""!
! CBR = $CHAR(125), "CLOSE BRACE 0110 1101 Y Y ""!
! TL = $CHAR(126), "TILDE 0110 1110 Y Y ""!
! RO = $CHAR(127); "RUBOUT 0110 1111 Y Y ""!
!""
!"" $CHAR(128) .. $CHAR(255) PRINT AS A SPACE ON THE TERMINAL ""!
!"" AND ARE UNASSIGNED ASCII GRAPHICS ""!
!""
!"" FOR A DETAILED DESCRIPTION OF CONTROL DATA 713 TERMINAL ""!
!"" CODES REFER TO - ""!
!""
!"" CONTROL DATA 713 CONVERSATIONAL DISPLAY TERMINAL ""!
!"" HARDWARE PROGRAMMING REFERENCE MANUAL ""!
!"" PUBLICATION NO. 62033400 (JUNE, 1971) ""!
!""
!"" OPR - INDICATES WHETHER A DISPLAY OPERATION IS PERFORMED (CURSOR MOVED) ""!
!"" CHR - INDICATES WHETHER A CHARACTER IS DISPLAYED ON THE TERMINAL SCREEN ""!
!"" GRP - SHOWS THE GRAPHIC ACTUALLY DISPLAYED ON THE TERMINAL SCREEN ""!
!""
!MODEND CHARACTER$SET$713;
!

```

APPENDIX H

SWL BNF (ALPHABETICAL)





20 <ADAPTABLE POINTER TO SEQUENCE> ::= <ADAPTABLE POINTER>

21 <ADAPTABLE POINTER TO STACK> ::= <ADAPTABLE POINTER>

22 <ADAPTABLE POINTER TO STRING> ::= <ADAPTABLE POINTER>

23 <ADAPTABLE POINTER> ::= -[[READ]]<ADAPTABLE TYPE>

24 <ADAPTABLE RECORD SPEC> ::=  
RECORD [<FIXED FIELDS> ,] <ADAPTABLE FIELD> RECENT

25 <ADAPTABLE RECORD TYPE IDENTIFIER> ::= <IDENTIFIER>

26 <ADAPTABLE RECORD> ::=  
[<PACKING>]<ADAPTABLE RECORD TYPE IDENTIFIER>  
\ [<PACKING>]<ADAPTABLE RECORD SPEC>

27 <ADAPTABLE SEQUENCE IDENTIFIER> ::= <IDENTIFIER>

28 <ADAPTABLE SEQUENCE> ::= <ADAPTABLE SEQUENCE IDENTIFIER>

29 <ADAPTABLE STACK IDENTIFIER> ::= <IDENTIFIER>

30 <ADAPTABLE STACK> ::= <ADAPTABLE STACK IDENTIFIER>  
\ STACK [#] OF <TYPE>

31 <ADAPTABLE STORAGE TYPE> ::= <ADAPTABLE STACK>  
\ <ADAPTABLE SEQUENCE>  
\ <ADAPTABLE HEAP>  
\ <ADAPTABLE STRING IDENTIFIER>

32 <ADAPTABLE STRING IDENTIFIER> ::= <IDENTIFIER>

33 <ADAPTABLE TYPE> ::= <ADAPTABLE AGGREGATE TYPE>  
\ <ADAPTABLE STORAGE TYPE>

34 <ADDING OPERATOR> ::= + \ - \ OR \ XOR \ UOR

35 <AGGREGATE TYPE> ::= <STRING TYPE>  
\ <ARRAY TYPE>  
\ <RECORD TYPE>

36 <ALIGNMENT> ::= ALIGNED

37 <ALLOCATE STATEMENT> ::=  
ALLOCATE <ALLOCATION DESIGNATOR> [ IN <HEAP VARIABLE> ]

38 <ALLOCATION DESIGNATOR> ::=  
<POINTER VARIABLE>  
\ <ADAPTABLE POINTER VARIABLE> : [<ADAPTABLE FIELD FIXER>  
[, <ADAPTABLE FIELD FIXER> ]]  
\ <POINTER TO BOUND VARIANT RECORD VARIABLE> :  
[ <TAG FIELD FIXER> [, <TAG FIELD FIXER> ] ]

```

39  <ALPHABET> ::= <LETTER>
        \<DIGIT>
        \<SPECIAL MARK>
        \<BLANKS>
        \<UNUSED MARK>

40  <ALTERNATIVE PARTS> ::= IF <EXPRESSION> THEN <STATEMENT LIST>
        [OR IF <EXPRESSION> THEN <STATEMENT LIST>]

41  <ANY INDEX> ::= <INDEX> \ <STARRED INDEX>

42  <ARRAY EXPRESSION> ::= <EXPRESSION>
        \<INDEFINITE VALUE CONSTRUCTOR>

43  <ARRAY SPEC> ::= ARRAY [<INDICES>] OF <COMPONENT TYPE>

44  <ARRAY TYPE IDENTIFIER> ::= <IDENTIFIER>

45  <ARRAY TYPE> ::= [<PACKING>]<ARRAY TYPE IDENTIFIER>
        \ [<PACKING>]<ARRAY SPEC>

46  <ARRAY VARIABLE> ::= <VARIABLE>

47  <ASCII CHARACTER> ::= <ALPHABET>\<UNPRINTABLE>

48  <ASSIGNMENT STATEMENT> ::= <VARIABLE> := <EXPRESSION>
        \<FUNCTION IDENTIFIER> := <EXPRESSION>
        \<SUCCESSOR ASSIGNMENT> "CF. 10.1.2"
        \<PREDECESSOR ASSIGNMENT>

49  <ATTRIBUTE> ::= <ACCESS ATTRIBUTE>
        \<STORAGE ATTRIBUTE>
        \<SCOPE ATTRIBUTE>

50  <ATTRIBUTES> ::= [<ATTRIBUTE>[,<ATTRIBUTE>]]

51  <BASE DESIGNATOR> ::= (<RADIX>)

52  <BASE TYPE> ::= <SCALAR TYPE>

53  <BASIC CONSTANT> ::= <SCALAR CONSTANT>
        \<COMPILE TIME VARIABLE> "CF. SECTION 12.1"
        \<REAL CONSTANT>
        \<POINTER CONSTANT>

54  <BASIC TYPE> ::= <SCALAR TYPE>
        \<REAL TYPE>
        \<POINTER TYPE>

55  <BEGIN STATEMENT> ::=
        BEGIN <DECLARATION LIST><STATEMENT LIST> END

56  <BLANKS> ::=

57  <BOOLEAN CONSTANT IDENTIFIER> ::= <IDENTIFIER>

```

58 <BOOLEAN CONSTANT> ::= FALSE\TRUE\<>BOOLEAN CONSTANT IDENTIFIER>

59 <BOOLEAN TYPE IDENTIFIER> ::= <IDENTIFIER>

60 <BOOLEAN TYPE> ::= BOOLEAN  
                   \<>BOOLEAN TYPE IDENTIFIER>

61 <BOUND VARIANT POINTER> ::= -[[READ]]<BOUND VARIANT RECORD TYPE>

62 <BOUND VARIANT RECORD TYPE> ::=  
           [<PACKING>] <BOUND VARIANT RECORD TYPE IDENTIFIER>  
           \[[<PACKING>] BOUND <VARIANT RECORD SPEC>  
           \[[<PACKING>] BOUND <VARIANT RECORD TYPE IDENTIFIER>

63 <CASE PART> ::= CASE <TAG FIELD SPEC> OF <VARIATIONS> CASEND

64 <CASE STATEMENT> ::= CASE <SELECTOR> OF <CASES>  
                           [ELSE <STATEMENT LIST>] CASEND

65 <CASES> ::= <A CASE>[;<A CASE>]

66 <CELL TYPE> ::= CELL

67 <CHARACTER CONSTANT IDENTIFIER> ::= <IDENTIFIER>

68 <CHARACTER CONSTANT> ::= '<ALPHABET>'  
                           \<>CHARACTER CONSTANT IDENTIFIER>  
                           \\$CHAR (<INTEGER>)

69 <CHARACTER TYPE IDENTIFIER> ::= <IDENTIFIER>

70 <CHARACTER TYPE> ::= CHAR\<>CHARACTER TYPE IDENTIFIER>

71 <COMMENTARY STRING> ::= "[<COMMENT CHARACTER>]"

72 <COMPILATION UNIT> ::= <MODULE DECLARATION> "CF. SECTION

73 <COMPONENT TYPE> ::= <TYPE>

74 <CONFORMITY> ::=  
           <TYPE IDENTIFIER> <TYPE TEST OPERATOR> <UNION VARIABLE>  
           \<>POINTER VARIABLE> <POINTER TYPE TEST OPERATOR>  
           <UNION VARIABLE>  
           \<>VARIABLE> <VALUE TYPE TEST OPERATOR> <UNION VARIABLE>

75 <CONSTANT DECLARATION> ::=  
           CONST [<CONSTANT SPEC> [, <CONSTANT SPEC>]]

76 <CONSTANT IDENTIFIER LIST> ::= <IDENTIFIER LIST>

77 <CONSTANT SPEC> ::=  
           <CONSTANT IDENTIFIER LIST> = <CONSTANT EXPRESSION>  
           \<>EMPTY>

78 <CONSTANT> ::= <BASIC CONSTANT>\<STRING CONSTANT>





15 <FILE SPEC> ::= <ACTUAL FILE NAME> [, <FILE ATTRIBUTES>]

16 <FILE TYPE> ::= LEGIBLE \ PRINT \ BINARY \ DIRECT

17 <FILE VARIABLE INITIALIZATION> ::=  
       := <FILE EXPRESSION>  
       \ := <INDEFINITE FILE VARIABLE CONSTRUCTOR>

18 <FILE VARIABLE SPEC> ::= <VARIABLE IDENTIFIERS> :  
       [<SCOPE OR STORAGE ATTRIBUTES>]  
       <FILE TYPE>      "CF. 4.8"  
       [<FILE VARIABLE INITIALIZATION>]  "CF. 6.7"

19 <FINAL VALUE> ::= <EXPRESSION>

20 <FIRST CHAR> ::= <POSITIVE INTEGER EXPRESSION>

21 <FIXED FIELD> ::= <FIELD SELECTORS> : [<ALIGNMENT>] <FIXED TYPE>

22 <FIXED FIELDS> ::= <FIXED FIELD> [, <FIXED FIELD>]

23 <FIXED OR VARIABLE BOUND TYPE> ::=  
       <BASIC TYPE> \ <STRUCTURED TYPE> \ <STORAGE TYPE>

24 <FIXED RECORD TYPE> ::= <INVARIANT RECORD TYPE>  
       \ <VARIANT RECORD TYPE>

25 <FOLLOWER> ::= <LETTER> \ <DIGIT> \ <#> \ <\$> \ <@>

26 <FOR LIST> ::=  
       <INITIAL VALUE> TO <FINAL VALUE> [BY <INCREMENT>]  
       \ <INITIAL VALUE> DOWNTO <FINAL VALUE> [BY <DECREMENT>]

27 <FOR STATEMENT> ::= FOR <CONTROL VARIABLE> := <FOR LIST> DO  
       <STATEMENT LIST> FOREND

28 <FORMAL PARAM LIST> ::= <IDENTIFIER LIST>

29 <FORMAL PARAM LIST> ::= <IDENTIFIER LIST>

30 <FORMAL POINTER> ::= <ADAPTABLE POINTER>  
       \ <POINTER TO CONTROL>  
       \ <BOUND VARIANT POINTER>

31 <FORMAL TYPE> ::= <ADAPTABLE TYPE>                  "CF. 4.5"  
       \ <CONTROL TYPE>                              "CF. 4.6"  
       \ <BOUND VARIANT RECORD TYPE>              "CF. 4.7"

32 <FREE STATEMENT> ::=  
       FREE <ALLOCATION DESIGNATOR> [IN <HEAP VARIABLE>]

33 <FUNCTION DESIGNATOR> ::=  
       <PROCEDURE REFERENCE> (<ACTUAL PARAMETER>  
       [, <ACTUAL PARAMETER>])  
       \ <PROCEDURE REFERENCE> ( )

```

134 <FUNCTION IDENTIFIER> ::= <PROCEDURE IDENTIFIER>
135 <GOTO STATEMENT> ::= GOTO [EXIT] <LABEL REFERENCE>
136 <HEAP TYPE> ::= HEAP (<SPACE>) "CF. 4.4.4"
137 <HEAP VARIABLE> ::= <VARIABLE>
138 <HEX DIGIT> ::= A\B\C\D\E\F
        \<DIGIT>
139 <IDENTIFIER LIST> ::= <IDENTIFIER>[,<IDENTIFIER>]
140 <IDENTIFIER> ::= <LETTER>[<FOLLOWER>]
141 <IF STATEMENT> ::=
        <ALTERNATIVE PARTS> IFEND
        \<ALTERNATIVE PARTS> ELSE <STATEMENT LIST> IFEND
142 <INCREMENT> ::= <EXPRESSION>
143 <INDEFINITE FILE VARIABLE CONSTRUCTOR> ::= [<FILE SPEC>]
144 <INDEFINITE VALUE CONSTRUCTOR> ::= [<VALUE ELEMENTS>]
145 <INDEX> ::= <SCALAR TYPE>
        \<SCALAR EXPRESSION> .. <SCALAR EXPRESSION>
146 <INDICES> ::= <INDEX>[,<INDEX>]
147 <INITIAL VALUE> ::= <EXPRESSION>
148 <INITIALIZATION> ::= := <EXPRESSION>
        \ := <INDEFINITE VALUE CONSTRUCTOR>
149 <INTEGER CONSTANT IDENTIFIER> ::= <IDENTIFIER>
150 <INTEGER CONSTANT> ::= <INTEGER>\<INTEGER CONSTANT IDENTIFIER>
151 <INTEGER TYPE IDENTIFIER> ::= <IDENTIFIER>
152 <INTEGER TYPE> ::= INTEGER\<INTEGER TYPE IDENTIFIER>
153 <INTEGER> ::= <DIGIT>[<DIGIT>]
        \<DIGIT>[<HEX DIGIT>] <BASE DESIGNATOR>
154 <INVARIANT RECORD SPEC> ::= RECORD <FIXED FIELDS> RECEND
155 <INVARIANT RECORD TYPE> ::=
        [<PACKING>] <INVARIANT RECORD TYPE IDENTIFIER>
        \ [<PACKING>] <INVARIANT RECORD SPEC>
156 <LABEL DECLARATION> ::= LABEL <LABEL>[, <LABEL>]

```

157 <LABEL OR PROC IDENTIFIER> ::= <LABEL>\<PROCEDURE IDENTIFIER>  
158 <LABEL REFERENCE> ::= <LABEL> \ <POINTER TO LABEL>--  
159 <LABEL TYPE> ::= LABEL  
160 <LABEL> ::= <IDENTIFIER>  
161 <LENGTH FIXER> ::= <SCALAR EXPRESSION>  
162 <LENGTH> ::= <POSITIVE INTEGER EXPRESSION>  
163 <LETTER> ::= A\B\C\D\E\F\G\H\I\J\K\L\M\N\O\P\Q\R\S\T\U\V\W\X\Y\Z  
164 <LOOP STATEMENT> ::= LOOP <STATEMENT LIST> LOOPEND  
165 <LOWER> ::= <CONSTANT SCALAR EXPRESSION>  
166 <MALIGNMENT> ::= MALIGNED (<OFFSET>[,<BASE>])  
167 <MEMBERS> ::= <TYPE LIST>  
168 <MODE> ::= READ[,<WRITE>] \ WRITE[,<READ>]  
169 <MODULE BODY> ::= <DECLARATION LIST>  
170 <MODULE DECLARATION> ::=  
MODULE [<MODULE IDENTIFIER>] [<(PRUNGS)>];  
<MODULE BODY>  
MODEND [<MODULE IDENTIFIER>]  
171 <MODULE IDENTIFIER> ::= <IDENTIFIER>  
172 <MULTIPLYING OPERATOR> ::= \* \ / \ MOD \ AND \ CAND  
173 <NEXT STATEMENT> ::=  
NEXT <ALLOCATION DESIGNATOR> IN <SEQUENCE VARIABLE>  
174 <NOT OPERATOR> ::= NOT  
175 <NTH> ::= <INTEGER EXPRESSION>  
176 <OBJECT TYPE> ::= <TYPE>  
177 <OLD OR NEW> ::= OLD \ NEW  
178 <ORDINAL CONSTANT IDENTIFIER LIST> ::=  
<ORDINAL CONSTANT IDENTIFIER>,<ORDINAL CONSTANT IDENTIFIER>  
[,<ORDINAL CONSTANT IDENTIFIER>]  
179 <ORDINAL CONSTANT IDENTIFIER> ::= <IDENTIFIER>  
180 <ORDINAL CONSTANT IDENTIFIER> ::= <IDENTIFIER>  
181 <ORDINAL CONSTANT> ::= <ORDINAL CONSTANT IDENTIFIER>

182 <ORDINAL TYPE IDENTIFIER> ::= <IDENTIFIER>

183 <ORDINAL TYPE> ::= (<ORDINAL CONSTANT IDENTIFIER LIST>  
 \<ORDINAL TYPE IDENTIFIER>

184 <PACKING ATTRIBUTES> ::= PACKED \ UNPACKED

185 <PACKING> ::= <PACKING ATTRIBUTES>

186 <PARAM SEGMENT> ::= <REFERENCE PARAMS>\<VALUE PARAMS>

187 <PARAMETER LIST> ::= (<PARAM SEGMENT>[;<PARAM SEGMENT>])

188 <PARENTAL TYPE> ::= <STORAGE TYPE>  
 \<AGGREGATE TYPE>

189 <POINTER CONFORMITY CASE STATEMENT> ::= CASE :-: <UNION VARIABLE>  
 OF <POINTER CONFORMITY CASES>[ELSE <STATEMENT LIST>] CASEND

190 <POINTER CONFORMITY CASES> ::=  
 <A POINTER CONFORMITY CASE>[;<A POINTER CONFORMITY CASE>]

191 <POINTER CONSTANT IDENTIFIER> ::= <IDENTIFIER>

192 <POINTER CONSTANT> ::= NIL

193 <POINTER REFERENCE> ::= <POINTER VARIABLE>  
 \<FUNCTION DESIGNATOR>

194 <POINTER TO CONTROL> ::= -<CONTROL TYPE>

195 <POINTER TO COPROC> ::= <FORMAL POINTER>

196 <POINTER TO COPROC> ::= <VARIABLE>

197 <POINTER TO FILE> ::= -<FILE TYPE>

198 <POINTER TO LABEL> ::= <FORMAL POINTER>

199 <POINTER TO PROCEDURE> ::= <FORMAL POINTER>

200 <POINTER TO TYPE> ::= -[[READ]]<FIXED OR VARIABLE BOUND TYPE>  
 \<POINTER TO FILE>

201 <POINTER TYPE SPECIFIER> ::= <POINTER VARIABLE>

202 <POINTER TYPE TEST OPERATOR> ::= :-:

203 <POINTER TYPE> ::= <DIRECT POINTER TYPE>  
 \<RELATIVE POINTER TYPE>

204 <POINTER VARIABLE> ::= <VARIABLE>

205 <POP STATEMENT> ::= POP <POINTER VARIABLE> ON <STACK VARIABLE>

206 <POSITION> ::= FIRST \ ASIS \ LAST

207 <POWER> ::= <FACTOR> \ <POWER> <EXPONENTIATION OPERATOR> <FACTOR>  
208 <PREDECESSOR ASSIGNMENT> ::= <SCALAR VARIABLE> :- <NTH>  
209 <PROC ATTRIBUTE> ::= XDCL \ REPDEP \ <SEGMENT IDENTIFIER>  
210 <PROC ATTRIBUTES> ::= <PROC ATTRIBUTE>[<PROC ATTRIBUTE>]  
211 <PROC BODY> ::= <DECLARATION LIST> <STATEMENT LIST>  
212 <PROC END> ::= PROCEND [<PROCEDURE IDENTIFIER> ]  
213 <PROC SPEC> ::= <PROCEDURE IDENTIFIER> <PROC TYPE SPEC>  
\ <PROCEDURE IDENTIFIER> <PROCEDURE TYPE>  
214 <PROC TYPE ATTRIBUTES> ::=  
<NULL CONSTRUCT (FOR EXPANSION PURPOSES)>  
215 <PROC TYPE SPEC> ::=  
[[<PROC TYPE ATTRIBUTES>]][<PARAMETER LIST>][<RETURN TYPE>]  
216 <PROCEDURE CALL STATEMENT> ::=  
<PROCEDURE REFERENCE> <ACTUAL PARAMETER LIST>  
217 <PROCEDURE DECLARATION> ::=  
PROC [ XREF ] <PROC SPEC>  
\ PROC[[<PROC ATTRIBUTES>]]<PROC SPEC>;<PROC BODY><PROC END>  
218 <PROCEDURE IDENTIFIER> ::= <IDENTIFIER>  
219 <PROCEDURE REFERENCE> ::= <PROCEDURE IDENTIFIER>  
\<POINTER TO PROCEDURE>-  
220 <PROCEDURE TYPE IDENTIFIER> ::= <IDENTIFIER>  
221 <PROCEDURE TYPE> ::= <PROCEDURE TYPE IDENTIFIER>  
\PROC <PROC TYPE SPEC>  
222 <PRONGS> ::= <IDENTIFIER LIST>  
223 <PUSH STATEMENT> ::= PUSH <POINTER VARIABLE> ON <STACK VARIABLE>  
\ PUSH <ALLOCATION DESIGNATOR>  
224 <RADIX> ::= 2 \ 4 \ 8 \ 10 \ 16  
225 <REAL CONSTANT IDENTIFIER> ::= <IDENTIFIER>  
226 <REAL CONSTANT> ::= <REAL NUMBER>\<REAL CONSTANT IDENTIFIER>  
227 <REAL NUMBER> ::= <UNSCALED NUMBER>  
\<SCALED NUMBER>  
228 <REAL TYPE IDENTIFIER> ::= <IDENTIFIER>  
229 <REAL TYPE> ::= REAL\<REAL TYPE IDENTIFIER>

230 <RECORD EXPRESSION> ::= <EXPRESSION>  
                                   \  
                                   <INDEFINITE VALUE CONSTRUCTOR>

231 <RECORD TYPE> ::= <FIXED RECORD TYPE>  
                                   \  
                                   <VARIABLE BOUND RECORD TYPE>

232 <RECORD VARIABLE> ::= <VARIABLE>

233 <REF TYPE> ::= <SWL TYPE>

234 <REF TYPE> ::= <SWL TYPE>

235 <REFERENCE PARAMS> ::=  
                   REF <FORMAL PARAM LIST> : [[ READ ]]<REF TYPE>

236 <RELATIONAL OPERATOR> ::= < \ <= \ > \ >= \ = \ /= \ IN

237 <RELATIVE POINTER TYPE> ::=  
                   RELI(<PARENTAL TYPE>)]-<OBJECT TYPE>

238 <REP SPEC> ::= REP <POSITIVE INTEGER EXPRESSION> OF

239 <REP TYPE> ::= <CELL TYPE>  
                   \  
                   <CRAMMED TYPE>

240 <REPEAT STATEMENT> ::= REPEAT <STATEMENT LIST> UNTIL <EXPRESSION>

241 <RESET STATEMENT> ::=  
                   RESET <SEQUENCE VARIABLE>[ TO <POINTER VARIABLE>]  
                   \  
                   RESET <STACK VARIABLE> [TO <POINTER VARIABLE>]  
                   \  
                   RESET <HEAP VARIABLE>

242 <RESUME STATEMENT> ::= RESUME (<COPROC REFERENCE>)

243 <RETURN STATEMENT> ::= RETURN [WHEN <EXPRESSION>]

244 <RETURN TYPE> ::= <BASIC TYPE>

245 <SCALAR CONSTANT> ::= <ORDINAL CONSTANT>  
                                   \  
                                   <BOOLEAN CONSTANT>  
                                   \  
                                   <INTEGER CONSTANT>  
                                   \  
                                   <CHARACTER CONSTANT>

246 <SCALAR TYPE> ::= <INTEGER TYPE>  
                                   \  
                                   <CHARACTER TYPE>  
                                   \  
                                   <ORDINAL TYPE>  
                                   \  
                                   <BOOLEAN TYPE>  
                                   \  
                                   <SUBRANGE TYPE>

247 <SCALED NUMBER> ::= <UNSCALED NUMBER> E[<SIGN>]<DIGIT>[<DIGIT>]

248 <SCOPE ATTRIBUTE> ::= XDCL \ XREF \ EXTERNAL

249 <SCOPE OR STORAGE ATTRIBUTE> ::=  
                   <SCOPE ATTRIBUTE> [,<STORAGE ATTRIBUTE>]  
                   \  
                   <STORAGE ATTRIBUTE> [,<SCOPE ATTRIBUTE>]

250 <SEGMENT DECLARATION> ::= SEGMENT <SEGMENTS>, <SEGMENTS>  
 251 <SEGMENT IDENTIFIER> ::= <IDENTIFIER>  
 252 <SEGMENT IDENTIFIERS> ::= <SEGMENT IDENTIFIER>  
                                   [, <SEGMENT IDENTIFIER>]  
 253 <SEGMENTS> ::= <SEGMENT IDENTIFIERS> : [[<ACCESS ATTRIBUTES>]]  
 254 <SELECTION SPEC> ::= <CONSTANT SCALAR EXPRESSION>  
                                   [..
 255 <SELECTION VALUE> ::= <CONSTANT SCALAR EXPRESSION>  
                                   [ .. <CONSTANT SCALAR EXPRESSION>]  
 256 <SELECTION VALUES> ::=  
                                   <SELECTION VALUE> [, <SELECTION VALUE>]  
 257 <SELECTION VALUES> ::= <SELECTION VALUE> [, <SELECTION VALUE>]  
 258 <SELECTOR> ::= <EXPRESSION>  
 259 <SEQUENCE TYPE> ::= SEQ (<SPACE>)  
 260 <SEQUENCE VARIABLE> ::= <VARIABLE>  
 261 <SET TYPE IDENTIFIER> ::= <IDENTIFIER>  
 262 <SET TYPE> ::= SET OF <BASE TYPE>  
                                   \  
                                   <SET TYPE IDENTIFIER>  
 263 <SIGN> ::= + \ -  
 264 <SIMPLE EXPRESSION> ::= <TERM> \ <SIGN><TERM>  
                                   \  
                                   <SIMPLE EXPRESSION>  
                                   <ADDING OPERATOR><TERM>  
 265 <SPACE> ::= <SPAN>[, <SPAN>]  
 266 <SPAN FIXER> ::= [ <SPAN> [, <SPAN> ] ]  
 267 <SPAN> ::= [REP <POSITIVE INTEGER EXPRESSION> OF]  
                                   <TYPE IDENTIFIER>  
 268 <SPECIAL MARK> ::= +\-\\*\./\.\;\:\\"'!  
                                   \  
                                   #\\$\%&@\?(\)\=\<\>  
 269 <STACK SIZE> ::= <INTEGER EXPRESSION>  
 270 <STACK TYPE> ::= STACK [ <STACK SIZE> ] OF <TYPE>  
 271 <STACK VARIABLE> ::= <VARIABLE>

272 <STAR FIXER> ::= <SCALAR EXPRESSION> .. <SCALAR EXPRESSION>  
273 <STAR> ::= \* \ \* : <SCALAR TYPE>  
274 <STARRED INDEX> ::= <STAR> \ <STARRED SUBRANGE>  
275 <STARRED LIST> ::=  
[<INDEX>,<INDEX>] <STARRED INDEX> [, <ANY INDEX>]  
276 <STARRED SUBRANGE> ::= \* .. <SCALAR EXPRESSION>  
\<SCALAR EXPRESSION> .. \*  
277 <STARRY SUBRANGE FIXER> ::= <SCALAR EXPRESSION>  
278 <STATEMENT LIST> ::= <STATEMENT>[;<STATEMENT>]  
279 <STATEMENT> ::= <UNLABELED STATEMENT>\<LABEL> : <STATEMENT>  
280 <STORAGE MANAGEMENT STATEMENT> ::= <PUSH STATEMENT>  
\<POP STATEMENT>  
\<NEXT STATEMENT>  
\<RESET STATEMENT>  
\<ALLOCATE STATEMENT>  
\<FREE STATEMENT>  
281 <STORAGE TYPE> ::= <STACK TYPE>  
\<SEQUENCE TYPE>  
\<HEAP TYPE>  
282 <STRING CONSTANT IDENTIFIER> ::= <IDENTIFIER>  
283 <STRING CONSTANT> ::= <STRING TERM> [CAT <STRING TERM>]  
284 <STRING TERM> ::= <CHARACTER CONSTANT>  
\<STRING CONSTANT IDENTIFIER>  
\'<ALPHABET> <ALPHABET> [<ALPHABET>]  
285 <STRING TYPE IDENTIFIER> ::= <IDENTIFIER>  
286 <STRING TYPE> ::= STRING (<LENGTH>) OF <CHARACTER TYPE>  
\<STRING TYPE IDENTIFIER>  
287 <STRING VARIABLE> ::= <VARIABLE>  
288 <STRUCTURED STATEMENT> ::= <BEGIN STATEMENT>  
\<IF STATEMENT>\<LOOP STATEMENT>  
\<WHILE STATEMENT>\<REPEAT STATEMENT>  
\<FOR STATEMENT>\<CASE STATEMENT>  
\<VALUE CONFORMITY CASE STATEMENT>  
\<POINTER CONFORMITY CASE STATEMENT>

289 <STRUCTURED TYPE> ::= <SET TYPE>  
                                  \<>UNION TYPE>  
                                  \<>AGGREGATE TYPE>

290 <SUBRANGE TYPE IDENTIFIER> ::= <IDENTIFIER>

291 <SUBRANGE TYPE> ::= <SUBRANGE TYPE IDENTIFIER>  
                                  \<>LOWER>..<>UPPER>

292 <SUBSCRIPT> ::= <SCALAR EXPRESSION>

293 <SUBSCRIPTED REFERENCE> ::= <ARRAY VARIABLE> [<SUBSCRIPTS>]

294 <SUBSCRIPTS> ::= <SUBSCRIPT>[,<SUBSCRIPT>]

295 <SUBSTRING LENGTH> ::= <POSITIVE INTEGER EXPRESSION>

296 <SUBSTRING REFERENCE> ::= <STRING VARIABLE>(<SUBSTRING SPEC>)

297 <SUBSTRING SPEC> ::= <FIRST CHAR>[,<SUBSTRING LENGTH>]

298 <SUCCESSOR ASSIGNMENT> ::= <SCALAR VARIABLE> :+ <NTH>

299 <SWL TYPE> ::= <DATA TYPE>  
                                  \<>FORMAL TYPE>

300 <TAG FIELD FIXER> ::= <SCALAR EXPRESSION>

301 <TAG FIELD SELECTOR> ::= <IDENTIFIER>

302 <TAG FIELD SPEC> ::=  
                  <TAG FIELD SELECTOR> : [<ALIGNMENT>] <TAG FIELD TYPE>

303 <TAG FIELD TYPE> ::= <SCALAR TYPE>

304 <TERM> ::= <POWER>\<>TERM><MULTIPLYING OPERATOR><POWER>

305 <TYPE DECLARATION> ::= TYPE [<TYPE SPEC>[,<TYPE SPEC>] ]

306 <TYPE IDENTIFIER LIST> ::= <IDENTIFIER LIST>

307 <TYPE IDENTIFIER> ::= <IDENTIFIER>

308 <TYPE LIST> ::= <TYPE> [, <TYPE>].

309 <TYPE SPEC> ::= <TYPE IDENTIFIER LIST> = <SWL TYPE> \ <EMPTY>

310 <TYPE TEST OPERATOR> ::= ::

311 <TYPE> ::= <FIXED OR VARIABLE BOUND TYPE> \ <FILE TYPE>

312 <UNION TYPE> ::= [<PACKING>] UNION (<MEMBERS>)

313 <UNION VARIABLE> ::= <VARIABLE>

314 <UNION VARIABLE> ::= <VARIABLE>

315 <UNLABELED STATEMENT> ::= <ASSIGNMENT STATEMENT>  
 \<STRUCTURED STATEMENT>[<LABEL>]  
 \<CONTROL STATEMENT>  
 \<STORAGE MANAGEMENT STATEMENT>  
 \<INPUT-OUTPUT STATEMENT>

316 <UNSCALED NUMBER> ::= <DIGIT>[<DIGIT>].<DIGIT>[<DIGIT>]

317 <UNUSED MARK> ::= &\ \[\]\%\@

318 <UPPER> ::= <CONSTANT SCALAR EXPRESSION>

319 <VAL TYPE> ::=  
 <TYPE> \ <ADAPTABLE TYPE> \ <BOUND VARIANT RECORD TYPE>

320 <VAL TYPE> ::=  
 <TYPE> \ <ADAPTABLE TYPE> \ <BOUND VARIANT RECORD TYPE>

321 <VALUE CONFORMITY CASE STATEMENT> ::=  
 CASE ::= <UNION VARIABLE> OF <VALUE CONFORMITY CASES>  
 [ELSE <STATEMENT LIST> ] CASEND

322 <VALUE CONFORMITY CASES> ::=  
 <A VALUE CONFORMITY CASE> [ ; <A VALUE CONFORMITY CASE> ]

323 <VALUE ELEMENT> ::= [<REP SPEC>]<EXPRESSION>  
 \ [<REP SPEC>]<INDEFINITE VALUE CONSTRUCTOR>  
 \ [<REP SPEC>] \*

324 <VALUE ELEMENTS> ::= <VALUE ELEMENT>[ , <VALUE ELEMENT> ]

325 <VALUE PARAMS> ::=  
 VAL <FORMAL PARAM LIST> : [ [ READ ] ] <VAL TYPE>

326 <VALUE PARAMS> ::=  
 VAL <FORMAL PARAM LIST> : [ [ READ ] ] <VAL TYPE>

327 <VALUE PARAMS> ::=  
 VAL <FORMAL PARAM LIST> : [ [ READ ] ] <VAL TYPE>

328 <VALUE TYPE SPECIFIER> ::= <VARIABLE>

329 <VALUE TYPE TEST OPERATOR> ::= ::=

330 <VARIABLE BOUND FIELD> ::=  
 <FIELD SELECTOR> : [ <ALIGNMENT> ] <VARIABLE BOUND TYPE>

331 <VARIABLE BOUND RECORD SPEC> ::=  
 RECORD [ <FIXED FIELDS> , ] <VARIABLE BOUND FIELD> RECEND

332 <VARIABLE BOUND RECORD TYPE> ::=  
     [<PACKING>]<VARIABLE BOUND RECORD TYPE IDENTIFIER>  
     \[<PACKING>]<VARIABLE BOUND RECORD SPEC>

333 <VARIABLE DECLARATION> ::=  
     VAR [<VARIABLE SPEC> [, <VARIABLE SPEC>]]  
     \ VAR [<FILE VARIABLE SPEC>[,<FILE VARIABLE SPEC>]]

334 <VARIABLE IDENTIFIER> ::= <IDENTIFIER>

335 <VARIABLE IDENTIFIERS> ::=  
     <VARIABLE IDENTIFIER> [, <VARIABLE IDENTIFIER>]

336 <VARIABLE REFERENCE> ::= <VARIABLE IDENTIFIER>  
     \<POINTER REFERENCE>-  
     \<SUBSTRING REFERENCE>  
     \<SUBSCRIPTED REFERENCE>  
     \<FIELD REFERENCE>

337 <VARIABLE SPEC> ::=  
     <TYPE>[INITIALIZATION]  
     \<EMPTY>

338 <VARIABLE> ::= <VARIABLE REFERENCE>

339 <VARIANT RECORD SPEC> ::=  
     RECORD [<FIXED FIELDS>,) <CASE PART> RECEND

340 <VARIANT RECORD SPEC> ::=  
     RECORD [<FIXED FIELDS>,) <CASE PART> RECEND

341 <VARIANT RECORD TYPE> ::=  
     [<PACKING>] <VARIANT RECORD TYPE IDENTIFIER>  
     \[<PACKING>] <VARIANT RECORD SPEC>

342 <VARIANT> ::= <FIXED FIELDS>  
     \[<FIXED FIELDS>,) <CASE PART>

343 <VARIATION> ::= =<SELECTION VALUES>= <VARIANT>

344 <VARIATIONS> ::= <VARIATION> [, <VARIATION>]

345 <WHILE STATEMENT> ::=  
     WHILE <EXPRESSION> DO <STATEMENT LIST> WHILEND

346 <WIDTH> ::= <INTEGER CONSTANT>



## LIST OF FIGURES

<u>CHAPTER 1: CONCEPTS OF SWL</u>	Figure	Page
THE COMPILATION UNIT	1.1	1-2
THE COMPILATION PROCESS	1.2	1-2
BEGIN-END BLOCK	1.3	1-3
NESTED BEGIN-END BLOCKS	1.4	1-4
BLOCK STRUCTURE	1.5	1-5
LOCAL VARIABLES	1.6	1-6
SCOPE OF IDENTIFIERS	1.7	1-7
SCOPE OF IDENTIFIERS (SWL)	1.8	1-8
IDENTIFIER CONFLICTS	1.9	1-9
A SIMPLE PROCEDURE BLOCK	1.10	1-10
NESTED PROCEDURE BLOCKS	1.11	1-10
FLOW OF EXECUTION DURING PROCEDURE CALLS	1.12	1-11
"  "  "  "  "  "	1.13	1-11
"  "  "  "  "  "	1.14	1-11
SHIELDING & SHARING VARIABLES	1.15	1-12
SCOPE OF VARIABLE IDENTIFIERS	1.16	1-12
PARAMETER PASSING	1.17	1-13
PARAMETER PASSING MECHANISMS	1.18	1-14
SIMPLE RECURSION	1.19	1-14
NOT-SO-SIMPLE RECURSION	1.20	1-15
FUNCTION CALL	1.21	1-15
A FORWARD & BACKWARD LINKED LIST	1.22	1-16
JUMP TABLE	1.23	1-16
THE SEQUENCE	1.24	1-18
THE STACK	1.25	1-18
THE HEAP	1.26	1-19
TYPE DEFINITION	1.27	1-20
VARIABLE DECLARATION	1.28	1-21
STATIC VARIABLE	1.29	1-21
EXAMPLE OF BNF	1.30	1-23
BNF INTEGERS	1.31	1-23
<u>CHAPTER 2: ELEMENTARY SWL</u>		
IDENTIFIERS	2.1	2-2
MODULE & MODEND STATEMENTS	2.2	2-3
COMMENTS	2.3	2-4
PROGRAM STRUCTURE	2.4	2-4
SCOPE OF VARIABLES	2.5	2-7
SUBRANGES WITH RADIX	2.6	2-8
CONSTANT DECLARATIONS	2.7	2-9
SIMPLE INPUT/OUTPUT	2.8	2-12
SIMPLE INPUT/OUTPUT (INTEGER)	2.9	2-12
CLASSES OF OPERATORS	2.10	2-13
ELEMENTARY COMPUTATION	2.11	2-15
CONVERSION FUNCTIONS	2.12	2-15
IF STATEMENT FLOWCHART	2.13	2-16

## CHAPTER 2: ELEMENTARY SWL (Continued)

	Figure	Page
IF STATEMENT SYNTAX	2.14	2-17
SHORT IF STATEMENT	2.15	2-17
IF STATEMENT SYNTAX	2.16	2-18
NESTED IF STATEMENTS FLOWCHART	2.17	2-18
NESTED IF STATEMENT SYNTAX	2.18	2-19
ORIF CLAUSE IN IF STATEMENT	2.19	2-19
CASES IN AN IF STATEMENT	2.20	2-19
CASE STATEMENT FLOWCHART	2.21	2-20
CASE STATEMENT SYNTAX	2.22	2-20
CASE STATEMENT SYNTAX	2.23	2-21
ELSE IN CASE STATEMENT	2.24	2-22
REPEAT STATEMENT FLOWCHART	2.25	2-23
REPEAT STATEMENT	2.26	2-23
REPEAT STATEMENT SYNTAX	2.27	2-24
WHILE STATEMENT FLOWCHART	2.28	2-25
WHILE STATEMENT	2.29	2-25
WHILE STATEMENT (FACTORIALS)	2.30	2-26
LOOP STATEMENT FLOWCHART	2.31	2-27
EXIT STATEMENT	2.32	2-27
LOOP STATEMENT	2.33	2-28
FOR STATEMENT FLOWCHART	2.34	2-29
FOR STATEMENT SYNTAX	2.35	2-30
FOR STATEMENT SYNTAX	2.36	2-31
REPETITIVE FOR STATEMENT	2.37	2-31
REPETITIVE REPEAT STATEMENT	2.38	2-32
REPETITIVE WHILE STATEMENT	2.39	2-32
REPETITIVE LOOP STATEMENT	2.40	2-32

## CHAPTER 3: SWL DATA STRUCTURES

TYPE DECLARATION	3.1	3-2
ORDINAL TYPE	3.2	3-3
ORDINALS	3.3	3-3
USE OF ORDINALS	3.4	3-3
CONVERTING ORDINALS TO INTEGERS	3.5	3-4
CONVERTING INTEGERS TO ORDINALS	3.6	3-5
TYPICAL ARRAY TYPE	3.7	3-5
USE OF ARRAYS	3.8	3-6
MULTI-DIMENSIONED ARRAYS	3.9	3-6
ARRAY REFERENCES	3.10	3-7
A BINARY TRUTH TABLE	3.11	3-7
A 12-BIT MEMORY WITH 12-BIT ADDRESSING	3.12	3-8
A CHARACTER CONVERSION TABLE	3.13	3-8
ARRAY OPERATIONS	3.14	3-9
STRING TYPE DECLARATION	3.15	3-9
BUILDING STRINGS	3.16	3-10
STRING SEARCHING	3.17	3-11
STRING SEARCH	3.18	3-12
FINDING ONE STRING IN ANOTHER	3.19	3-13
SUB-STRING SEARCH	3.20	3-13

### CHAPTER 3: SWL DATA STRUCTURES

	Figure	Page
POINTER TYPES	3.21	3-15
POINTER REFERENCES	3.22	3-16
POINTER SYMBOLS	3.23	3-17
USE OF POINTERS	3.24	3-17
THE CONCEPTUAL RECORD	3.25	3-18
RECORD SYNTAX	3.26	3-19
A FORWARD LINKED LIST	3.27	3-20
LINKED LIST IN SWL	3.28	3-20
DECLARATION OF SETS	3.29	3-21
MAKING FULL SET	3.30	3-22
SET OPERATION EXAMPLES	3.31	3-23
STORAGE ALLOCATION	3.32	3-24, 25
FREEING A LINKED LIST	3.33	3-25
SEQUENCE SYNTAX	3.34	3-26
XDCL and XREF ATTRIBUTES	3.35	3-28
PACKED DATA	3.36	3-29
VARIABLE INITIALIZATION	3.37	3-29
ARRAY INITIALIZATION	3.38	3-30
RECORD INITIALIZATION	3.39	3-30
RECORD INITIALIZATION VALUES	3.40	3-30
SET INITIALIZATION	3.40.1	3-31
STRING INITIALIZATION	3.40.2	3-31
READING INPUT FROM DATA FILE	3.41	3-32
READING INPUT FROM TERMINAL	3.42	3-33

### CHAPTER 4: ADVANCED SWL

SIMPLE PROCEDURE DECLARATION & CALL	4.1	4-2
ACTUAL and FORMAL PARAMETERS	4.2	4-3
PROCEDURE WITH PARAMETERS	4.3	4-4
FUNCTION DECLARATION	4.4	4-5
NESTED PROCS	4.5	4-7
XDCL and XREF PROCEDURES	4.6	4-8
SIMPLE SUMMATION FUNCTION	4.7	4-9
ADAPTABLE SUMMATION FUNCTION	4.8	4-10
ADAPTABLE ARRAY SPECIFICATIONS	4.9	4-11
ADAPTABLE STRING SPECIFICATION	4.10	4-11
ARRAY OF UNION OF POINTERS	4.11	4-12
UNION SYNTAX	4.12	4-12
UNIONS IN PROCEDURES	4.13	4-13
UNION OPERATORS	4.14	4-13
VALUE TYPE TESTING OPERATOR	4.15	4-14
VALUE CONFORMITY CASE SYNTAX	4.16	4-15
VARIANT RECORD CONCEPT	4.17	4-16
SWL VARIANT RECORD SYNTAX	4.18	4-16
VARIANT RECORD LAYOUT	4.19	4-17
VARIANT RECORD INITIALIZATION	4.20	4-17
LABELS	4.21	4-18
NON-LABELED EXIT	4.22	4-19
LABELED EXIT	4.23	4-19
NON-LABELED CYCLE STATEMENT	4.24	4-20
LABELED CYCLE STATEMENT	4.25	4-21
USE OF RETURN STATEMENT	4.26	4-22

## CHAPTER 4: ADVANCED SWL (Continued)

	Figure	Page
FILE DECLARATIONS	4.27	4-23
FILE STRUCTURE	4.28	4-24
GET-FILE I/O	4.29	4-24
PUT-FILE	4.30	4-25
REPDEP FUNCTION #/SIZE	4.31	4-26
REPDEP CELL AND #/LOC FUNCTION	4.32	4-26
STANDARD FUNCTIONS	4.33	4-27

## CHAPTER 5: STRUCTURED PROGRAMMING AND SWL

AN UNSTRUCTURED PROGRAM	5.1	5-2
A STRUCTURED PROGRAM	5.2	5-3
PROGRAM OVERVIEW	5.3	5-5
TESTSQUARE PROCEDURE	5.4	5-6
SETQUEEN PROCEDURE	5.5	5-6
STEPWISE REFINEMENT	5.6	5-7,8
MODIFICATION TO PROGRAM OF FIGURE 5.6	5.7	5-9

## CHAPTER 6: SWL PROGRAMMING TECHNIQUES AND CONVENTIONS

DIFFICULT TO MAINTAIN VAR DECLARATION	6.1	6-2
EASY TO MAINTAIN VAR DECLARATION	6.2	6-2
PROGRAM CLARITY	6.3	6-3

## CHAPTER 7: PERFORMANCE MEASUREMENT & PREDICTION

PERFORMANCE MEASUREMENT SYSTEM DESCRIPTION	7.1	7-4
SAMPLE RANGE vs RESOLUTION	7.2	7-5
SWLSMP INTERFACE FUNCTION	7.3	7-7,8
SUBMIT FILE FOR SAMPLING	7.4	7-9
TYPICAL LOAD MAP	7.5	7-11
MEMORY LAYOUT DERIVED FROM LOADER MAP	7.6	7-12
SOURCE PROGRAM TO BE SAMPLED	7.7	7-14
ACTUAL ADDRESS COMPUTATION	7.8	7-15
SAMPLE OUTPUT - 1st PART	7.9	7-16
SAMPLE OUTPUT - 2nd PART	7.10	7-19
REPETITIVE STATEMENT RESULTS	7.11	7-20
PERFORMANCE IMPROVEMENT CYCLE	7.12	7-21

# I N D E X

- Adaptable Array, 4-11
- ADAPTABLE Types, 4-9, 4-11
  - Arrays, 4-11
  - Strings, 4-11
  - Records, 4-11
  - Stacks, 4-11
  - Heaps, 4-11
  - Sequences, 4-11
- Allocate Statement, 1-19, 3-24
- Arrays, 3-5, 3-6
- Array References, 3-7
- Assignment Operator ("="), 2-13
- Automatic Variables, 1-21, 2-8
  
- Backus Naur Form, 1-23, H-1-H-17
- BEGIN-END BLOCK, 1-3, 1-4, 1-6
- BEGIN Statement, 1-3, 1-5
- Block Structure, 1-3, 1-5
  
- CASE Statement, 2-16, 2-20, 2-21, 4-15
- CASEND Statement, 2-20
- CDC-713 Character Codes, G-1-G-4
- CELL Data Type, 4-25
- Classes of Operators, 2-13
  - NOT Operator, 2-13
  - Multiplicative Operator, 2-13
  - Additive Operator, 2-13
  - Relational Operator, 2-13
- CLOSE Statement, 4-25
- Comments, 2-3, 2-4
- Compilation Listing, F-1
- Compilation Unit, 1-2
- Compile-Time Options, 4-28
- CONST Statement (Constant Declaration), 2-9
- Constant, 2-9
- Conversion Functions, 2-15, 2-16
- CYCLE Statement, 4-18, 4-20, 4-22
  
- Data Representation Dependent Features  
 ([REPEDEP]), 4-25
  
- END-OF-FILE Condition, 2-24
- END Statement, 1-3, 1-5
- Error List, B-1-B-4
- EXIT Statement, 2-27, 2-28, 4-18, 4-20
  
- FOR Statement, 2-16, 2-28-2-31, 4-20
- FREE Statement, 1-19, 3-24
  
- Functions, 1-15, 4-5
  - Declarations, 4-6
  - Call, 4-6
- GET Statement, 4-24
- GLOBAL Variables, 1-6, 1-7, 1-9, 1-12
- GOTO Statement, 4-22
  
- Heap, 1-19
  
- Identifiers, 2-2
- IF Statement, 2-16, 2-18
- IFEND Statement, 2-17
- Input/Output, 2-10
  
- Jump Table, 1-16
  
- LABEL Statement, 4-18
- Language Summary, C-1, C-2
- Linked Lists, 1-16
- Load Map, 7-11
- #LOC Function, 4-26
- Local Variables, 1-6, 1-7, 1-9, 1-12
- LOOP Statement, 2-16-2-28
- #LOWERBOUND Function, 4-10
  
- Module
  - (declaration), 1-2, 2-7
- MODULE-MODEND Statements, 2-3
  
- Nested IF Statement, 2-18
- Nested Procedures and Functions, 4-7
- NEXT Statement, 3-26
  
- OPEN Statement, 4-23
- Operator Classes, 2-13
- ORDINALS, 3-2-3-5
- ORIF Clause, 2-19
  
- Packed Variables, 3-28
- Parameter Passing (Functions), 1-15, 4-3
- Parameter Passing (Procedures), 1-13, 1-14
- Pointers (Declaration), 3-15
- Pointer References, 3-16
- Pointer Symbols, 3-17
- Pointer Variable
  - (forward and backward pointers), 1-16
- #PRED FUNCTION, 4-27
- Predefined Variable Types, 1-20
  - Integer, 2-5, 12-6
  - Character, 2-5, 12-6

Predefined Variable Types (Cont)  
   Real, 2-5, 12-6  
   Boolean, 2-5, 12-6  
 Procedures, 1-11  
 Procedure Actual Parameters, 4-3  
 Procedure Call, 4-2, 4-3  
 Procedure Formal Parameters, 4-3  
 PROC-END (Procedure) BLOCK, 1-3, 1-9,  
   1-10, 4-2  
 PROC-PROCEND Statements, 2-4, 2-5  
 PUT Statement, 4-25  
  
 READ Statement, 2-10, 2-12  
 Records, 3-18  
 RECORD-RECORD, 3-18  
 Record References, 3-19  
 Recursive Calls (Procedures) 1-14, 1-15  
 REPEAT Statement, 2-16, 2-22, 2-23  
 Repetitive Statement Comparison, 2-31, 2-32  
 Repetitive Statements, 2-22  
 Replaceable Blocks, 1-5  
 Reserved Words, A-1  
 RESET Statement, 3-26  
 RETURN Statement, 4-21  
 REWIND (Input) Statement, 3-32, 3-33, 4-25  
  
 SCOPE Attributes, 3-27  
 Scope of Identifiers, 1-6, 1-7, 1-8  
 Sequence Storage Management Scheme, 1-18  
 Sequence Type, 3-26  
 SETS, 3-21  
**Set Initialization, 3-31**  
 Sharing Variables, 1-6, 1-12  
 Shielding Variables, 1-6, 1-12  
 #SIZE Function, 4-26  
 SMP (Data Collection Program), 7-3-7-10  
 SMP Output, 7-16-7-21  
 Stacks, 1-18  
 Standard Functions, 4-27  
   \$REAL, 4-27  
   \$INTEGER, 4-27  
   \$CHAR, 4-27  
   \$STRING, 4-27  
   #STRLENGTH, 4-27  
   #LOWERBOUND, 4-27  
   #UPPERBOUND, 4-27  
   #EOF, 4-27  
   #LOC, 4-27  
   #SIZE, 4-27  
   #STRINGREP, 4-27  
   #SUCC, 4-27  
   #PRED, 4-27  
   #ABS, 4-27  
   Static Variables, 1-21, 1-22  
   Storage Attributes (Variables) (Automatic,  
     STATIC), 3-27  
   Storage Management, 1-18, 3-24  
   String Conversion, 3-14  
**String Initialization, 3-31**  
   String Referencing, 3-10  
   \$STRINGREP Function, 3-14  
   Strings, 3-9  
   #STRLENGTH (Adaptable String Variable  
     Name), 4-11  
   Structured Program, 5-3, 5-4  
   Structured Statements, 2-16  
   Substrings, 3-12  
   SUCCESSOR (#SUCC) and PREDESSOR (#PRED)  
     Functions, 4-27  
  
 Terminal Session, E-1-E-7  
 Type Conformity, 3-31  
 Type Declaration, 1-20  
  
 UNION Operators, 4-13  
 UNION Type, 4-12  
 Universal Heap, 3-21  
 #UPPERBOUND Function, 4-10  
 User-Defined Variable Types, 1-20  
   Ordinal, 1-20  
   Subrange, 1-20  
   Pointer, 1-20  
   Structured, 1-20  
  
 Value Type Testing, 4-14  
 VAR Statement, 1-5, 1-6, 2-5, 3-2  
 Variables, 1-21  
 Variable Attributes, 3-27  
 Variable Declaration, 1-20, 3-2, 6-2  
 Variable Initialization, 3-29  
 Variable Type, 1-20  
 Variable Type Checking, 1-20  
 Variable  
   Type Declarations (TYPE), 3-2  
 Variant LONG FORM, 4-17  
 Variant Records, 4-16  
 Variant SHORT FORM, 4-17  
  
 WHILE Statement, 2-16, 2-24-2-26  
 WRITE Statement, 2-10, 2-11, 2-12  
  
 XDCL (Declared External) Attribute  
   (Procedures and Functions), 2-4, 3-27, 4-7  
 XREF Attribute (Procedures and Functions)  
   3-27, 4-7