Burroughs

B 1700 Systems User Programming Language (UPL)

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A.

December 1973

170

Burroughs

B 1700 Systems User Programming Language (UPL)

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. December 1973 1067170

AA494075

Any comments or suggestions regarding this publication should be forwarded to Publications Department, Technical Information Organization, TIO-West, Burroughs Corporation, 9451 Telstar Avenue, El Monte, California 91731.

Table of Contents

Table of Contents (cont)

List of Illustrations

List of Tables

INTRODUCTION

The User Programming Language (UPL) has been developed specifically for writing the system software for the B 1700 Series. UPL is a high-level, problem-oriented language that allows sophisticated computer programs to be written with relative ease.

This reference manual has been designed and written for experienced programmers. It can be used to learn the language, however, if the programmer is familiar with bit-manipulation concepts and language-independent principles of programming.

UPL is a compiler-level language that increases programmer productivity and solves complex problems. The resultant system software reflects this increased productivity.

SECTION 1

LANGUAGE CHARACTERISTICS

GENERAL.

The type of problems to be solved by UPL has required a series of functions and constructs that differ significantly from most other problem-oriented languages. A few of these differences are as follows:

a. Powerful bit and character-string functions.
b. Binary-only arithmetic functions.
c. No JUMP or GO TO instruction.
d. Re-entrant programs.
e. Recursive procedures (subroutines).
f. Scope of data-names contained within procedures.
g. Dynamic storage allocation for data-names at execution time.
h. Single-pass compilation.

All programs that are written in UPL source language must be processed by another program, the UPL Compiler. The compiler transforms the source statements into a virtual machine form called S-code. The S-code is then executed interpretatively by a set of micro-instruction routines (firmware).

UPL PROPERTIES.

A UPL Program has a distinct pattern or format that specifies the relative locations of the two statement types, declarative and executable. Declarations provide the information that is needed to allocate storage or link together various elements of a program. Executable statements specify the functions or transformations to be performed upon the contents of storage.

Statements are composed of symbols that, in turn, are composed of letters, digits, and special characters. Symbol strings are called operands, operators, or control functions. The UPL syntax is concerned with the correct creation of symbol strings and the relative placement of the strings to form declarative or executable statements.

UPL PROGRAM FORMAT.

UPL Programs are segmented into logical subdivisions called procedures. Each procedure begins with a head statement and terminates with an end statement. Procedures have a definite relationship to other procedures within a program, either side-by-side (parallel) or subordinate (nested). This ordering inherently defines the scope of each procedure and the range over which a procedure can call or be called.

PROCEDURE FORMAT.

All procedures have a rigid internal structure. The procedure structure is as follows: first, data-name declarations; second, all nested procedures; and last, all executable statements. The structure of nested procedures must be exactly the same.

METALANGUAGE.

A metalanguage is a language that is used to describe other languages. Symbols in the metalanguage are called metalinguistic symbols. Meta-

linguistic symbols are used in forming metalinguistic formulas. The formulas define the rules of allowable sequences of characters and symbols in the language being described. Thus, a set of metalinguistic formulas defines the syntax of a language.

The following set of metalinguistic symbols is used throughout this manual to describe the UPL syntax.

KEY WORDS.
All underlined, upper case words are key words within a statement and are required when the functions they are part of are utilized. Their omission causes error conditions at compilation time. Examples of key words are as follows:

$$\text{IF} \quad \left\{ \begin{array}{l} \text{literal} \\ \text{data-name} \\ \text{expression} \end{array} \right\} \quad \text{THEN statement; [ELSE statement;]}$$

The key words are IF, THEN, and ELSE. (Refer to paragraph on brackets for exception.)

LOWER CASE WORDS.
All lower case words represent generic terms that must be supplied in the specified format position by the programmer. Literal, data-name, expression, and statement are generic terms in the preceding example.

BRACES.
When words or phrases are enclosed in braces ({ }), a choice of one of the entries must be made. With reference to the preceding example, one of the items (literal, data-name, or expression) must be included in the statement.

BRACKETS.
Words and phrases enclosed in brackets ([ ]) represent optional portions of a statement. In terms of the preceding example, the [ELSE statement;] can be included in the statement as an option; otherwise, it is omitted.

CONSECUTIVE PERIODS.
The presence of an ellipsis (. . .) within any format indicates that the syntax immediately preceding the notation can be successively repeated, depending upon the requirements of problem solving.

PERIOD.
The period, or dot, is used only to concatenate parts of data-names, for example,

```
WORK.SPACE.ONE.
```

TYPE (LENGTH).
The type (length) phrase always represents the following syntactical notation:

{ FIXED
  CHARACTER (length)
  BIT (length) }

Any mark or symbol in a metalinguistic formula that is not one of the metalinguistic symbols denotes itself. The juxtaposition of symbols in the formula denotes juxtaposition of the elements in the language being described.

Metalinguistic formulas give an accurate and detailed description of the legal sequences of symbols within a language. They do not, however, assign meaning or indicate the events performed by the statements in the target language. Such a description is called the semantics of a language. Therefore, for each syntactical description of an SDL construct within this manual, a semantics portion also appears.

BASIC_SYMBOLS.
The UPL character set is composed of the following:

a. The upper case letters A through Z are used to form names and strings.

b. The digits 0 through 9 are used to form numbers in literals and in strings.

c. The arithmetic operators + (addition), - (subtraction), * (multiplication), and / (division) provide mathematical capabilities.

d. The relational operators > or GTR (greater than), < or LSS (less than), = or EQL (equal to), ≠ or NEQ (not equal to), ≥ or GEQ (greater than or equal to), and ≤ or LEQ (less than or equal to) provide comparison capabilities.

e. The logical operators are AND, OR, EXOR (exclusive OR), and NOT (negation).

f. The functional operators CAT (concatenation), MOD (results in the remainder of a divide), and := or ← (replacement) provide additional functions that are required.

g. The following punctuation defines the function of each symbol that is used in UPL.

Symbol	Definition	Use
.	Period or dot	Concatenation within data names
,	Comma	Separator for items
;	Semicolon	Delimiter for statements
(	Left parenthesis	Enclose parameter lists

Symbol	Definition	Use
)	Right parenthesis	Enclose parameter lists
"	Quotation mark	Left and right character delimiter
#	Pound sign	Left and right define text string delimiter
	Space or blank	Data-name delimiter
←	Arrow	Assignment or replacement (delete left) operator symbol
:←	Colon, arrow	Replacement (delete right) operator symbol
@	At sign	String delimiter
:=	Colon, equal	Assignment or replacement (delete left) operator symbol
::=	Colon, colon, equal	Replacement (delete right) operator symbol
%	Percent sign	Remainder of card is a comment
/*	Slash, asterisk	Beginning of comment
*/	Asterisk, slash	End of comment
?	Question mark	In column 1 of a 96-column card, indicates an MCP control card
?	Invalid punch	In column 1 of an 80-column card, indicates an MCP control card
$	Dollar sign	In column 1, indicates a compiler control card
&	Ampersand	In column 1, denotes conditional source code inclusion control card

RESERVED WORDS.

UPL contains a set of character strings, called reserved words, with pre-assigned meanings. Three classes of reserved words are defined.

Class I reserved words have pre-assigned meanings throughout the program, for example, DECLARE, PROCEDURE, DO, END. Incorrect usage of a class I reserved word results in a syntax error.

Class II reserved words can be re-assigned meanings. They then lose their original meanings for the duration or scope of their new assignment, for example, CONV, DECIMAL, LENGTH. Re-assignment of a class II reserved word results in a warning message, but no syntax error.

Class III reserved words have pre-assigned meanings only within some input/output statements, for example, DISK, LOCK, PRINTER, TAPE. Incorrect usage of a class III reserved word within a specific UPL statement results in a syntax error. The words, when used in input/output statements, must appear as shown in the syntax and cannot be DEFINEd. The usage of a class III reserved word in any other portion of the program is considered as a separate and distinct usage and does not result in a syntax error.

A helpful list of all classes of reserved words is given in appendixes A, B, and C.

LANGUAGE STATEMENT TYPES.

There are seven types of statements in UPL. Their names, forms, and a brief description of their functions are as follows:

Name	Form	Function
Assignment	Data-name := expression;	Performs calculations and assigns a value to a data-name
Declaration	DECLARE data-name attributes; DECLARE data-name REMAPS data-name attribute list;	Reserves space for, and assigns attributes to, data-names
Conditional	IF expression THEN statement; ELSE statement;	Controls the execution of individual statements or groups of statements
Control	DO FOREVER name; statement; statement; . . .	Iterates, groups, or transfers control to sets of statements

Name	Form	Function
	statement; END name; CASE expression; statement; statement; . . . statement; END CASE; RETURN; UNDO;	
Procedure	A procedure is a set of statements.	Defines a subset of the program to be used as a subroutine
Simple	BUMP data-name; DECREMENT data-name;	Performs some simple function on a data-name
Compiler information	DEFINE FORWARD PROCEDURE SEGMENT	Assists the programmer in preparing and compiling a program

FUNCTIONS.

There are several functions in UPL; they have been incorporated into UPL to facilitate ease of use and speed of execution. Examples of a few functions and a brief description of them are as follows:

Name	Function
SUBSTR	Addresses substring within a character field
SUBBIT	Addresses substring with a bit field
LENGTH	Obtains the length of a substring
CONV	Converts between data types
MOD	Obtains the remainder of a divide operation
CAT	Concatenates substrings

Name	Function
BINARY	Converts from printable decimal to binary
DECIMAL	Converts from binary to printable decimal

The use of each function is described in section 8.

SECTION 2

BASIC CONCEPTS

GENERAL.

UPL has a number of basic concepts that a programmer must understand in order to fully utilize the language. These concepts are explained in the following paragraphs.

DATA CONCEPTS.

A data-name is the symbolic name associated with a memory space. The data-name is DECLAREd with a set of attributes describing the space and how it is to be manipulated. An occurrence of a data-name references the contents of the memory space with its associated attributes.

There are three declarable classifications of data in UPL.

FIXED DATA TYPE.

The type FIXED data format is a signed 24-bit field. It is the primary computational form in the language. The most significant bit is the sign. A 0 denotes a positive number; a 1 denotes a negative number. The remaining 23 bits are the value in binary. If the number is negative, the value is in the complement notation of 2. The maximum and minimum values are 2, raised to the 23rd power -1 (8,388,607 in decimal) and -2, raised to the 23rd power (-8,388,608). All calculations are in binary, and any overflow beyond the largest expressible value is ignored.

BIT DATA TYPE.

The type BIT data format assumes a string of binary digits that can be manipulated or interpreted in any manner the programmer chooses. Bit-strings may be declared from 1 to 65,535 bits in length. Bit-literals are also available in 1-, 2-, 3-, and 4-bit groups. Type BIT data can be used in arithmetic operations and is always considered a 24-bit positive number; that is, the maximum and minimum values are 2 raised to the 24th power -1 (16,777,215 in decimal) and 0 (zero). If the data item is greater than 24 bits, the high order positions are converted to 0's during arithmetic operations. Comparison operations are performed on the whole bit-string in a right-to-left manner with leading 0's padded on the shorter string; that is, 110 compares less than 1000.

The data-name that is declared a bit-string manipulates the whole string. Substrings of the data-name can be manipulated with the SUBBIT function.

CHARACTER DATA TYPE.

The type CHARACTER data format is an 8-bit-string grouping defined as standard EBCDIC. All input/output (I/O) peripheral devices, excluding data-communication devices, send and/or receive in the CHARACTER format. Arithmetic operations can be performed with CHARACTER data; however, the binary value of the CHARACTER bit-string is its binary arithmetic value. That is, a 0 character from a peripheral device has a binary value of 240 (11110000). Also, only the least significant (right-most) 24 bits of a CHARACTER data-name are used in arithmetic operations.

All high-order bits are converted to 0's (zeros). Comparisons are of two classes:

a. Character-to-character, which is compared left-to-right with EBCDIC spaces (hexadecimal 40) padded to the right of the shorter string.

b. Character to any other data format, which is compared right-to-left with binary 0's padded to the left of the shorter string.

Substrings of character-strings may be addressed with the SUBSTR function.

DATA TYPE CONVERSION.
Several conversion functions in UPL can transform from one data type to another. They are CONV, BINARY, and DECIMAL.

The same memory space can be declared as being of different data types each with unique data-names by the use of the REMAPS option and/or the structured options in the DECLARE statement.

ARRAYS.
An array is a repetitive set of data-elements. The data-name becomes the name of the whole array, and individual elements in the array are addressed by subscripting the data-name. Arrays are single dimensional; that is, they allow only one value in the subscript. The array declaration (*) can be used with all three data-types.

DATA STORAGE ALLOCATION.
Data storage allocation is divided into two distinct periods of time. The first is the compiler-time encounter of a DECLARE statement in which the compiler generates the code that performs the run-time allocations. The second occurs at object run-time when the actual storage allocations are performed upon entrance into a procedure and then only for those data-names DECLAREd in that procedure. When the procedure is RETURNed from, that is, exited, the physical memory locations again become available for allocation to any data-name that may be DECLAREd in the next procedure to be entered. That is, storage allocations and de-allocations are performed during entrance to or exit from each procedure at object run-time. The same physical memory space can, therefore, be used many times during the execution of a program (dynamic storage utilization).

DUPLICATE DATA-NAMES.
It is possible to have duplicate data-names in UPL that are not a compile-time error. This is true whenever the duplicate data-names are DECLAREd in different procedures. The occurrence of duplicate data-names within one procedure is an error and results in a compiler error message.

Duplicate data-names do not interfere because they exist only within the scope of their procedures. The case occurs, however, when the procedure that contains the duplicate data-name is nested within the

procedure that contains the first occurrence of the data-name. The language resolves this conflict by referencing the most recent occurrence of the data-name over the scope of the nested procedure. When this procedure returns control, the original data-name is again available.

ASSIGNMENT

The assignment operation moves the contents of one data-name, called the source field, into the memory space of another data-name, called the destination field. Alignment, truncation, or padding is performed during the assignment operation and is controlled by the type and length attributes of the data-names involved.

The type attribute divides alignment control into two cases. The first case is character-to-character, which aligns the data-names on their left-most or high-order characters. The assignment is, then, performed in a left-to-right order until one of the fields is exhausted. If the destination field is shorter, the operation ends. If the source field is shorter, the destination field is padded, on its right, with space characters (hexadecimal 40).

The second case includes every other possible combination of data types. The fields are aligned on their right-most or low-order bits, and the assignment proceeds from right-to-left until one of the fields is exhausted. If the destination field is shorter, the operation ends. If the source field is shorter, the destination field is padded with binary 0's.

REPLACEMENT

The replacement operator is similar to the assignment operator because both transfer data into a data-name and perform alignment, truncation, and padding during the transfer.

Differences between assignment and replacement operators involve use of a machine register and completion or incompletion of the source language statement. The assignment operator clears the register as it moves the contents into memory and, thus, ends a statement. The replacement operator, however, does not clear the register as it moves its contents into memory. The value or address remaining in the register must be used in further operations until the assignment operation is executed or until the register contents are no longer needed. This case occurs with an expression evaluation and is used as a conditional indicator as in the IF or CASE statements. For example:

```
X := A + (B := C);
```

The := symbol between data-names B and C is a replacement operator because a value or address of a value must be available to be added to data-name A. The := symbol between data-names X and A, however, is an assignment operator because nothing remains after the operation to be combined with another term; that is, the source language statement is completed.

Exactly what remains behind or, more formally, not deleted after a replacement operation is under control of the programmer. That is, the programmer can choose to leave behind either side of the replacement operation, which is then combined with the next term in the expression.

There are, therefore, two forms of the replacement operator: the delete left form, := , and the delete right form, ::= . Normal usage is the delete left form. Usage of the delete right part is often convenient in parameter passing to procedures. For example, if the procedure SQF requires a parameter of six characters and the programmer would like to use the procedure with a 4-character data-name, X4, the programmer can declare or use an existing 6-character data-name, X6, and do a replace, delete right part (::=) in the procedure call. For example,

```
SQF (X6 ::= X4);
```

SINGLE-PASS COMPILER.

The UPL Compiler is designed to pass the source language only once. This design has several ramifications in the program structure of the source language.

A rigid sequence of statement types is required in order to guarantee that the proper information is available to the compiler at the proper time. All DECLARE statements, for example, must appear within each procedure before any executable statements occur in that procedure. Also, each procedure must begin and end, in the view of the compiler, before any executable statement in some other procedure can reference it. Procedures, therefore, have a range or scope over which they can be referenced and are active. This scope is dependent on when a procedure occurs, in the view of the compiler; in what procedure nest it occurs; and how deep it is in the nest.

A special statement, the FORWARD statement, is available to resolve the problem of forward referencing a procedure that has not yet been seen by the compiler.

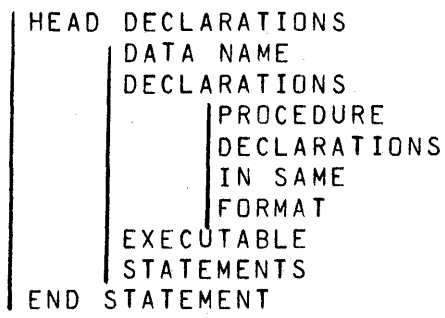
Procedures cannot overlap; however, they can be nested. Procedures also can be side-by-side (parallel), not nested, within an outer procedure, each of which can itself contain more nested or side-by-side procedures.

PROCEDURES.

A procedure is the basic structural element in UPL. It contains local data and the executable code for manipulating that data. It can also communicate values and/or addresses, that is, parameters, to and from other procedures. A procedure, in addition, can manipulate any data in the other procedures that are within scope (global).

A procedure is divided into the following five parts: the head declaration, the data declarations, the nested procedure declarations, the executable statements, and the END statement, in that order.

For example:



Procedures are analogous to subroutines in other languages. They execute repetitively the same set of statements by manipulating a different set of parameters on each invocation.

The outer-most level of code, that is, the level not imbedded in any procedure, conforms to the format of procedures except that it has no head declaration and cannot, therefore, be invoked by a procedure. This outer-most, procedure-like structure is referred to as the global level or lexicographical (lexic) level 0 (zero). Each subsequent procedure has a lexic level-number greater than 0.

PROCEDURE INVOCATION.
A procedure is invoked or called by use of the procedure-name in an executable statement. If the procedure has parameters, they must appear following the procedure-name and be surrounded by parentheses. The parameters associated with each call of a procedure are the actual parameters, and those within the procedure are the formal parameters.

PARAMETERS TO PROCEDURES.
A parameters-to-procedures transfer is considered a special case of storage allocation at procedure invocation time.

The procedure head declaration contains a data-name for every parameter that is passed. This data-name is called the formal parameter name and is used within the procedure to reference the passed information. A FORMAL declaration statement must follow the procedure head declaration and specify the type and attributes of the parameters.

At compile-time, code is generated to allocate memory space for these parameters. At run-time, the memory space is allocated and the actual parameter is loaded. Run-time comparisons of the actual parameter types and lengths against the formal types and lengths are performed only if the $ FORMAL.CHECK compiler option is specified. A mismatch causes program termination. A VARYING option in the FORMAL statement is available. It results in the use of the type and length of the actual parameter as the type and length of the corresponding formal parameter on each invocation of the procedure.

ACTUAL PARAMETERS.
Actual parameters are the data-names or the values contained in the data-names that are passed to procedures. They are matched in a left-

to-right order with the formal parameters in the procedure head declaration. They also must agree in number with the number of formal parameters.

FORMAL PARAMETERS.
Formal parameters are the symbolic data-names that are used in a procedure to reference and manipulate the actual parameters that are passed.

PASS-BY-VALUE.
The value of a data-name can be passed to a procedure. When the procedure is invoked, a copy of the value is loaded into the local memory space associated with the corresponding formal-name.

A parameter always passes-by-value as the result of an arithmetic operation upon the parameter or by surrounding the parameter with another set of parentheses, for example,

```
SQRF (A + B, A * B, (A));
```

The procedure SQRF has three actual parameters that all pass-by-value.

Array elements can be passed-by-value or by-name. The whole array can only be passed-by-name. The formal parameter declaration within the procedure can use the FORMAL.VALUE option; the parameter then always passes-by-value.

A pass-by-value has no lexic level restrictions. An inner procedure can invoke an outer procedure that is in scope and pass to it the value of a data-name that is out of the scope of the outer procedure.

PASS-BY-NAME.
The memory-space address of a data-name used as an actual parameter can be passed to a procedure. Local space is not allocated within the procedure; instead, the formal parameter is loaded with the address as passed. The procedure can, then, manipulate the original value as referenced by the data-name. The procedure can alter this original value. This occurs only if the formal parameter is the object of a replacement or assignment operation during the execution of the procedure.

The pass-by-name occurs whenever the actual parameter is not involved in an arithmetic operation or surrounded by an extra set of parentheses, for example,

```
SQRF (A, B, C + 2);
```

The actual parameters A and B are passed-by-name and the C + 2 is passed-by-value.

Notice that a pass-by-name has no lexic level restrictions. A procedure can pass the name of a data-name at a lower lexic level (higher number) to an outer procedure at a higher lexic level (lower number). The outer procedure eventually executes a RETURN to the

inner procedure; however, it had access to the memory space of a non-existent data-name and could have altered its value. Notice that data-name space continues to exist for every procedure invoked until that procedure executes a RETURN statement. This is true regardless of the lexic level or levels of any procedures invoked while a procedure exists. Only those data-names that are within scope, however, may be referenced. For example, a duplicate data-name at a lower lexic level inhibits the outer data-name during the time its procedure is active.

PROCEDURE TYPES.
Two types of procedures exist in UPL: one that performs a set of statements and then returns control, called a regular procedure, and another that performs a set of statements and returns a value when it returns control, called a function procedure.

REGULAR PROCEDURE.
A regular procedure call is a complete executable statement. That is, its name and parameter list are followed by a semicolon.

For example,

```
SQROOT (ABS);
```

is a regular procedure call.

A regular procedure may appear as a statement in the IF statement.

```
IF ABS NEQ ZERO THEN SQROOT (ABS);
```

After the regular procedure executes a RETURN statement, control passes to the next sequential statement following the call.

A regular procedure can communicate data by referencing global data-names.

FUNCTION PROCEDURE.
A function procedure call is considered a value and is used within expressions. It can be the source of a replacement operator (on the right of the replacement sign); or it can be operated upon by any of the arithmetic or logical functions, that is, added to or compared to.

When a function procedure RETURNs a value, it is used in place of the function call within the expression. The function procedure head statement contains a type-length attribute, and the RETURN statement contains an expression. The value of the expression must have this type-length attribute and is then passed back into the invoking expression.

For example:

```
PRICE := COST.SQFT * SQRT (LGTH, WDTH);
```

A value is RETURNed from the SQRT (LGTH,WDTH) function call and multiplied by data-name COST.SQFT.

And:

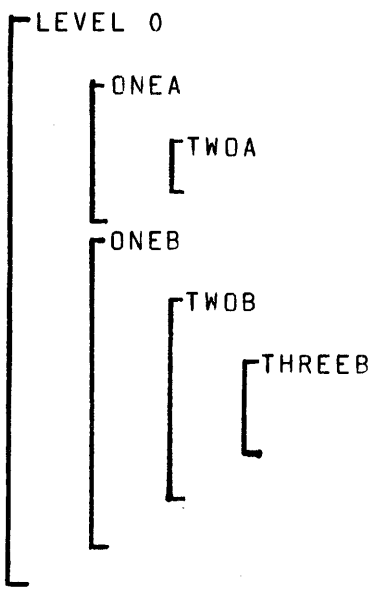
```
IF SUBSTR(MSG.IN,0,3) EQL "YES" THEN ... ;
                                 ELSE ... ;
```

The procedure MSG.IN accepts an input message from the SPO and returns a character-string that is tested for the first three characters equal to the word YES.

A number of functions exist in UPL that may be used as if the programmer has written function procedures for them. These functions are described in section 8.

LEXICOGRAPHIC LEVEL.

A lexicographic level is the compile-time relationship of each procedure to the outer level of the program. This outer level is referred to as lexicographic level 0 (zero). All other procedures are nested within it and are assigned a lexicographic level-number representing their depth of nesting from level 0. Thus, in the following example,



procedures ONEA and ONEB are at lexicographic level 1; TWOA and TWOB are at level 2; and procedure THREEB is at level 3.

The maximum lexicographic level is 15. That is, nested procedures can not exceed 15 levels in depth. There is no limit, however, to the number of procedures that can occur on any level or in any procedure.

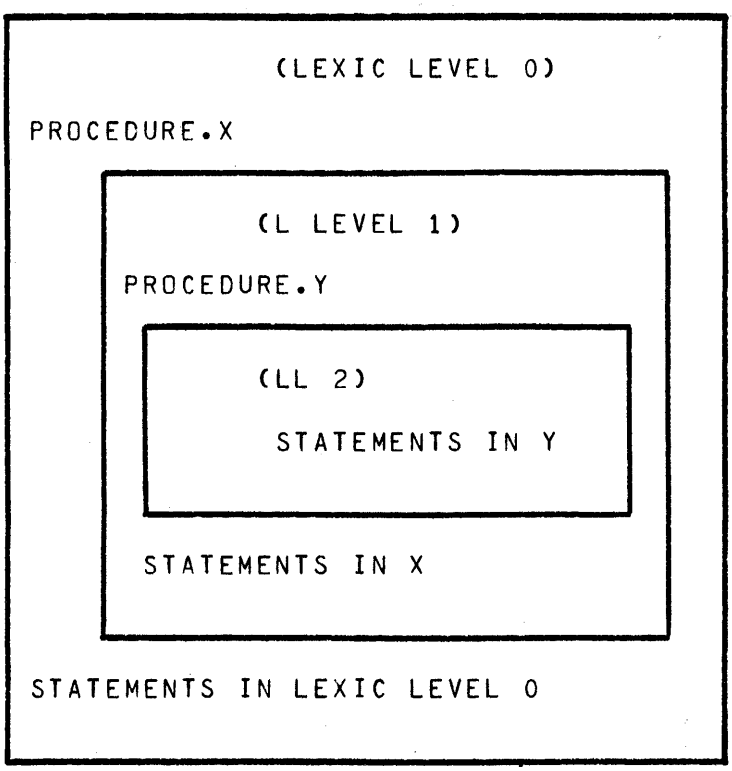
The naming of a procedure should not be confused with the procedure itself. The name of a procedure exists at some lexic level and denotes that a procedure is beginning with the next source language statement. This next source statement exists within the named procedure and is one

lexic level lower than the name of the procedure. That is, the name of a procedure exists one lexic level above the procedure that it names. This separation of the name and the procedure being named has significance in the concept of scope.

SCOPE.

Scope is the range within a program over which a data-name or procedure-name can be referenced. The scope of a name is a direct result of the lexicographic level of procedures and the storage allocation techniques employed.

Before a procedure is invoked, the names declared within the procedure do not exist yet and cannot be referenced. After the procedure is invoked, the names within the procedure can be referenced. The format of procedures ensures that only those statements contained within this procedure or in global procedures are within scope. That is, executable statements within a procedure can reference the names declared in this procedure or in any outer procedure. For example:



The statements in lexic level 0 may reference PROCEDURE.X but not PROCEDURE.Y because PROCEDURE.X has not been invoked and, therefore, the name of PROCEDURE.Y does not exist yet.

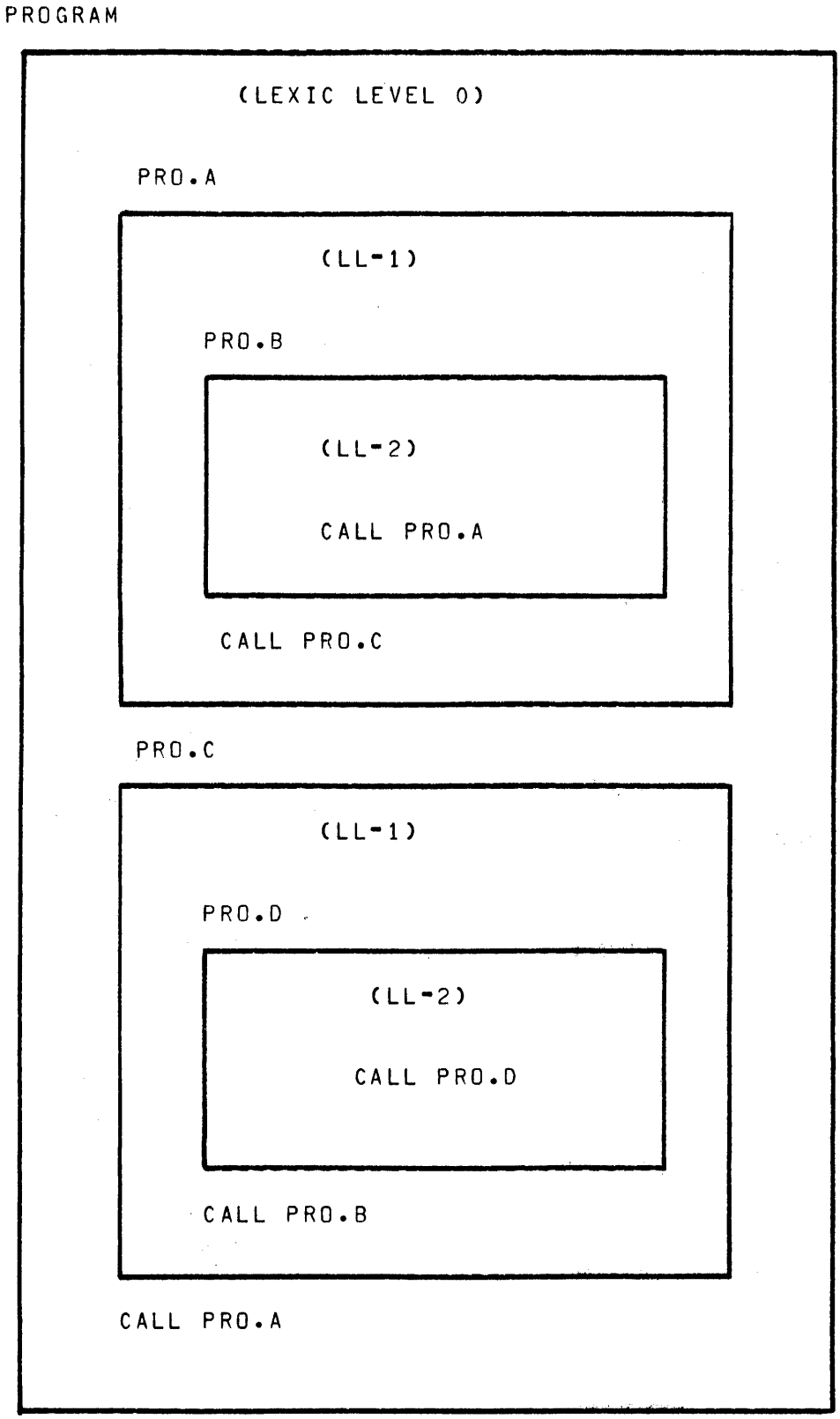
The statements in PROCEDURE.X can, however, reference PROCEDURE.Y because the name of PROCEDURE.Y becomes available when PROCEDURE.X is invoked.

The statements in PROCEDURE.X also can reference any names of data or procedures that are declared on lexic level 0. This implies that PROCEDURE.X can invoke itself, which is true; all procedures in UPL are recursive. Any difficulties encountered with duplicate names whether they be from recursive procedure invocations or just duplicate names within a nested procedure are resolved by the allocation of new space for the most recent occurrence of the duplicate name. Notice that the name of the whole program, the name of the lexic level 0 procedure, is outside of lexic level 0 and cannot be referenced from within the program. That is, the program cannot be called recursively because its name is not within scope.

Statements in PROCEDURE.Y can reference names within PROCEDURE.Y and PROCEDURE.X and on lexic level 0. That is, in this program, the statements in PROCEDURE.Y can reference any data-name and any procedure-name. Notice that the name PROCEDURE.Y exists when PROCEDURE.X is entered. That is, the name of a procedure is made available in its outer procedure.

Several procedures can have the same lexic level number by occurring at the same depth from lexic level 0. The relationships that can exist between such procedures depends upon the relationship of the nests in which they appear.

Procedures that have a common procedure one lexic level up can invoke each other. Procedures that do not have this attribute can not invoke each other. This condition is called not being within scope. For example:


PROGRAM
(LEXIC LEVEL 0)
PRO.A
(LL-1)
PRO.B
(LL-2)
CALL PRO.A
CALL PRO.C
PRO.C
(LL-1)
PRO.D
(LL-2)
CALL PRO.D
CALL PRO.B
CALL PRO.A

The procedures PRO.A and PRO.C are both on lexic level 1 (LL-1) and have a common procedure (lexic level 0 is considered a procedure in the above example) that is one lexic level higher. They can both, therefore, contain executable statements that invoke the other. The single-pass characteristic of UPL, however, requires a FORWARD statement before this is allowed.

Procedures PRO.B and PRO.D also have a common lexic level number; however, they are not nested in a common, immediately preceding procedure and cannot, therefore, reference each other.

A question now arises: What happens if PRO.B invokes PRO.A and then PRO.A invokes PRO.C? Can PRO.C now reference PRO.B or any of its data-names? It is, after all, still active because it has not yet executed a RETURN statement.

The answer is no. At **no time** can any statement in PRO.C reference PRO.B or any data-name in PRO.B. The programmer, however, never loses control because this run-time nest of invocations is eventually unwound. Each procedure **must** eventually execute a RETURN statement and pass control back to its invoking procedure, thus unwinding the nest.

The scope of a data-name is similar to the scope of a procedure-name. A data-name can be referenced by any executable statement within the procedure in which it is declared. It also can be referenced by any active procedure nested within its declaring procedure. It can **not** be referenced by any procedure with a lower lexic level number. Its value or name can, however, be passed as a parameter to such procedures. (Refer to page 2-6.)

For example:

```
PROGRAM

DECLARE (AA, ABC) FIXED;

PROCEDURE PRO.A;

    DECLARE BB FIXED;

    PROCEDURE PRO.B;

        DECLARE CC FIXED;

        ABC := AA + BB + CC;

    ABC := AA + BB;

PROCEDURE PRO.C

    DECLARE DD FIXED;

    ABC := AA + DD;

ABC := AA;
```

The data-names ABC and AA can be seen by (are within the scope of) all procedures. The data-name BB can be seen only by procedures PRO.A and PRO.B. That is, PRO.C and the outer level 0 code cannot reference data-names BB or CC.

SECTION 3

EXPRESSIONS

GENERAL.

Expressions are the operational portions of statements. If a statement is analogous to a sentence, then expressions are the words and phrases within a sentence. All operational functions, that is, comparison, arithmetic, etc., take place within expressions except the assignment and the regular procedure-call functions.

The format of an expression is similar to the format of an algebraic expression. Infix notation is used, and parentheses can be used, to group the order of evaluation. Each variable also can contain a prefix unary operator.

An expression is defined to be recursive and can, therefore, contain as many variables and operators as are required to produce the desired result.

Expressions are evaluated by performing the indicated operations in a left-to-right order. The sequence in which the operations are performed is determined by rules of precedence. (An operator precedence chart is given on page 3-5.) When operators have the same precedence, the sequence of operation is determined by the order of their appearance, from left-to-right. Parentheses can be used to modify the normal hierarchical sequence of execution. An expression within the parentheses is evaluated, and this value is then used in subsequent operations.

FORMAT OF AN EXPRESSION.

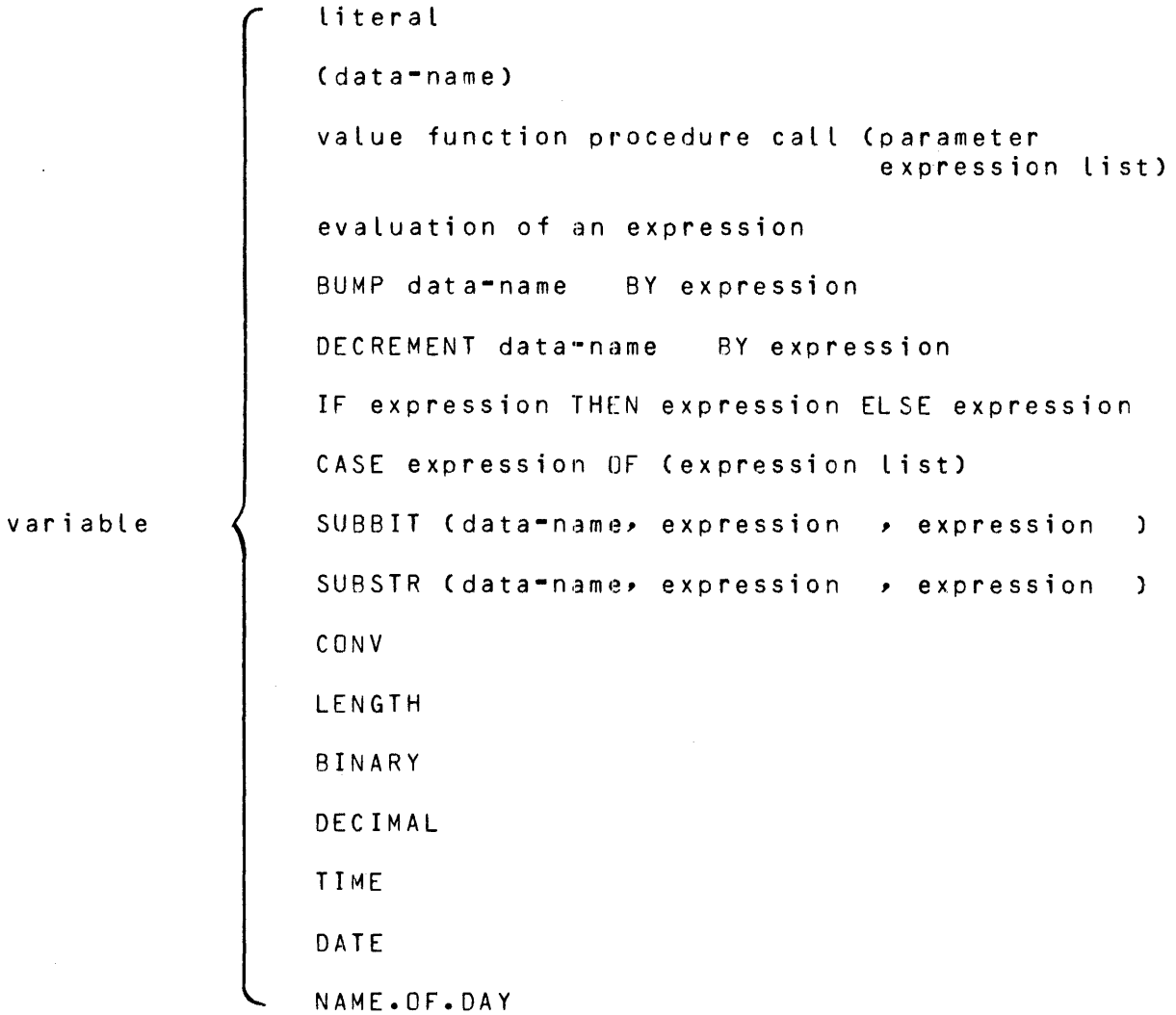
The expression syntax is as follows:

```
[ UNARY    ] { data-name } [ [ OPERATOR [ UNARY    ] { data-name } ] ... ]
[ OPERATOR ] { variable  }                [ OPERATOR ] { variable  }
```

The elements of the syntax are defined as follows:

data-name

- simple data-name
- array data-name
- substring { SUBBIT / SUBSTR } (data-name, expression, expression)
- BUMP data-name BY expression
- DECREMENT data-name BY expression
- IF expression THEN data-name ELSE data-name
- CASE expression OF (data-name list)



DATA-NAMES.
A data-name is a memory address with a type and length attribute. It is considered to have a value when used in an expression unless it appears to the left of a replacement or assignment operator, in which case it is the receiver of a value. A data-name differs from a variable because of this last characteristic. That is, a variable is always a value and can never be the receiver of other values. It can only be operationally combined with other values to produce a new value.

SIMPLE DATA-NAME.
A simple data-name is the data-name as given in a DECLARE statement.

ARRAY DATA-NAME.
An array data-name must be subscripted. The subscript that is enclosed in parentheses can be an arbitrary complex expression. For example:

Expression	Comments
X(I)	X is the array data-name, and the binary value of the simple data-name I is the array element being accessed. The first array element is number 0.
X (BUMP I BY 2)	Every other element of the array X is to be accessed.

SUBSTRINGS OF DATA-NAMES.
Two substring functions are available: the SUBBIT, to reference a string of bits within a data-name, and the SUBSTR, to reference a string of characters. Both functions require the data-name, the displacement starting locations, and the length of the substring. For example:

Expression	Comments
SUBSTR (X, 2, 1)	The third character in the data-name X is referenced.
SUBBIT (A, 7)	It references the eighth bit through the end of data-name A. The default option on the length is through the end of the data-name.

The arguments of the substring function can be expressions and, therefore, contain substring functions. For example:

Expression	Comments
SUBBIT (SUBSTR (X, 2, 1), 0, 2)	This references the first two (left-most) bits of the third character in data-name X.

The BUMP, CASE, DECREMENT, and IF expressions are discussed in detail in section 6.

VARIABLE.
A variable is a value within an expression. It can be explicit, such as a literal, or implicit, such as the sum of two data-names.

LITERAL.
A literal is a constant whose value is used in the expression as it appears. Three types of literals correspond to the three possible data-types.

a. The first is the fixed or constant literal. Fixed literals are signed number values. For example:

```
 43
-17
```

b. The second is the character or character-string literal. Character literals must be enclosed by quote (") symbols. They are 8-bit groups (EBCDIC). If a quote sign is desired in the character literal, two adjacent quote marks are required. The maximum number of characters is 256. For example:

```
"YES"
"     "
"QUOTE(""")"
```

c. The third type is the bit or bit-string literal. Bit literals are available in groupings of 1 (binary), 2 (quartal), 3 (octal), or 4 (hexadecimal). The bit literal is enclosed by @ signs and contains a grouping indicator enclosed in parentheses. For example:

```
@(1) 11011011@      binary
@(2) 01201122@      quartal
@(3) 0123577@       octal
@(4) 0148ACF@       hexadecimal
```

If the grouping indicator is omitted, 4 (hexadecimal) is assumed. The grouping can be changed within the literal. For example,

```
@(1) 101 (2) 210 (3) 765 (4) F@
```

is a 22-bit literal.

DATA-NAME VALUES.
Any data-name that is enclosed in an extra set of parentheses has only its value used in the expression. This is often useful when passing parameters to procedures.

VALUE FUNCTION PROCEDURE CALL.
A function procedure call is the use of the function-name in an expression. It passes control to any function procedure written by the programmer or to one of the intrinsic functions. It always results in a value.

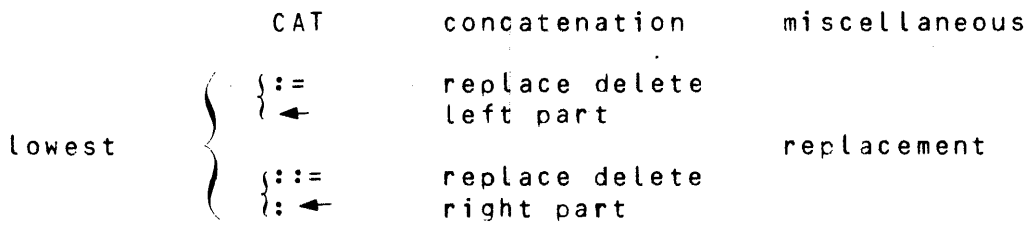
EVALUATION OF AN EXPRESSION.
The evaluation of an expression is the combining of any two or more operands with an operator, for example, A + B. The evaluation always results in a value.

SUBSTRINGS.
The bit substring function SUBBIT is used to isolate a set of bits from a data-name. The character substring function performs the same operation, but considers its data as type CHARACTER. When a substring function is used as a variable, it returns a value.

The BINARY, CONVERT, DATE, DECIMAL, LENGTH, NAME.OF.DAY, and TIME are described in section 8.

OPERATOR PRECEDENCE IN EXPRESSIONS.
Operator precedence in expressions is as follows:

	Operator	Function	Type
highest	:= or ←	replace delete left part	replacement
	::= or :←	replace delete right part	
	+	plus	unary
	-	minus	
	+	addition	arithmetic
	-	subtraction	
	*	multiplication	
	/	division	
	MOD	remainder	
	= EQL	equal	relational
	≠ NEQ	not equal	
	> GTR	greater than	
	≥ GEQ	greater than or equal to	
	< LSS	less than	
	≤ LEQ	less than or equal to	
	NOT	not	logical
	AND	and	
	OR	or	
	EXOR	exclusive-or	



The replacement operator is lower than any operator to its right, but higher than any operator to its left. Also, the replacement operator is distinct from the assignment operator in that the replacement must be within an expression while the assignment defines a statement.

EXPRESSION_TYPES.
Expressions can be divided into several types with corresponding properties. They all, however, can be combined or imbedded in any order and are collectively referred to as **expression** in the syntax.

ARITHMETIC EXPRESSIONS.
The arithmetic operations are as follows:

a. Addition (+).
b. Subtraction (-).
c. Multiplication (*).
d. Division (/).
e. Remainder of division (MOD).

The arithmetic operations are always performed on the low-order or right-most 24-bits of data-names, and the calculations are in binary regardless of the data-name declared type. Intermediate results are held in UPL machine registers, leaving the original contents of all data-names unaltered. An assignment or replacement operator is required to move the results of a calculation into a data-name.

Data-names can have preceding (post-fix) unary operators to the right of the arithmetic operators that are between (in-fix) the data-names.

FIXED ARITHMETIC EXPRESSIONS.
Arithmetic operations in which both operands are of type FIXED always produce a type FIXED result. The maximum numbers are +2 raised to the 23rd power -1 (+8,388,607) and -2 raised to the 23rd power (-8,388,608). Values in excess of these limits have their high-order bit ignored. Negative values are expressed in the complement notation of 2.

NON-FIXED ARITHMETIC EXPRESSIONS.
All arithmetic operations in which one or both of the operands are **not** of type FIXED are handled in a 24-bit mode.

The operands are considered as 24-bit unsigned positive numbers. The maximum value is 2 raised to the 24th power (16,777,216). If an expression containing non-FIXED data results in a value greater than 2 raised to the 23rd power -1, and it is then assigned or replaced into

a FIXED data-name, the result is a negative FIXED value. Use of the unary minus sign (-) with a non-FIXED data-name converts all 24-bits to 2's complement notation. This value is not a negative number, but it can add to other data-names to produce the equivalent of subtraction.

RELATIONAL OR CONDITIONAL EXPRESSIONS.
A relational expression determines the truth or falsity of a relationship between data-names. The result of a relation test can be used to control an IF statement or as an expression in further operations.

For example:

```
IF A EQL B THEN STMT.TRUE;
           ELSE STMT.FALSE;
```

The relationship of equality is used to choose between the two statements in the IF statement.

The relational test results in a single bit indicator, called the conditional-bit. Truth of the test sets this conditional-bit equal to a 1 (one), and falsity sets it equal to a 0 (zero).

Other operations utilizing the conditional-bit are allowed. For example, X := A EQL B; assigns X a value of binary 1 if A equals B or a value of 0 if A is unequal to B. The statement X := (A LSS B) + (A GTR B); results in X = 0 only if A equals B.

Relational comparisons are divided into three types:

a. A character-to-character comparison is left-aligned and bit-by-bit. Therefore, B is greater than A because B is hexadecimal C2 and A is hexadecimal C1. Shorter fields are padded with space characters (hexadecimal 40).

b. A fixed-to-fixed comparison is left-aligned and a true algebraic compare. That is, a -1 (negative one) is less than a 0 (zero).

c. All other combinations of data-types are right-aligned and bit-by-bit.

Any combination of operators can be included within a relational expression. For example,

```
IF  ((A NEQ B) OR ((A * B) NEQ ZERO) + 1)
        THEN ...;
        ELSE ...;
```

The relational operators are as follows:

Symbol	Mnemonic	Meaning
=	EQL	equal to
≠	NEQ	not equal to
>	GTR	greater than
<	LSS	less than
≥	GEQ	greater than or equal to
≤	LEQ	less than or equal to

LOGICAL EXPRESSIONS.
The logical operators perform a right-adjusted bit-by-bit combination of their operands. All data types are operated upon in the same manner for their full length. The shorter of the two is padded on its left with binary 0's. Notice that character-to-character logical operations are performed in the same manner as all other combinations of data types.

Table 3-1 illustrates the use of each logical operator.

Table 3-1

Logical Operator Usage

Variables	Bit Configuration			
IF X =	0	0	1	1
IF Y =	0	1	0	1
NOT X =	1	1	0	0
NOT Y =	1	0	1	0
X AND Y =	0	0	0	1
X OR Y =	0	1	1	1
X EXOR Y =	0	1	1	0

"Not" is considered a unary operator and may appear adjacent to any other operator including itself.

FUNCTION EXPRESSIONS.
A function expression is a call upon a user-written function procedure or upon one of the UPL-supplied functions as described in section 8.

In either case, the function returns a term to be combined with other terms to produce a result. The terms can be values or addresses.

Values can be operated upon directly as if they were literals to produce a result, while addresses point to memory space that can then be the source or destination of a value. That is, address-generating functions differ from value functions because they can be the object of an assignment or replacement operator.

For example:

```
A := LENGTH (x) ;
```

The LENGTH function returns a value equal to the number of units of X, and this value is assigned to the data-name. The LENGTH function can never appear to the left of an assignment or replacement operator.

The SUBBIT and SUBSTR functions, however, are address-generating functions and can appear on either side of an assignment or replacement operator. That is, SUBSTR(CARD,0,8); := SUBSTR(CARD,72,8); moves the right-most eight characters of the data-name CARD to the left-most eight character positions.

NOTE
Data-name CARD is assumed to
be an 80-character card image.

The VALUE-generating functions in UPL are as follows:

SUBBIT	BINARY
SUBSTR	TIME
CONVERT	DAY
LENGTH	NAME OF DAY
DECIMAL	BUMP
CASE	DECREMENT
IF...THEN...ELSE	

The ADDRESS-generating functions in UPL are as follows:

```
SUBBIT
SUBSTR
IF...THEN address generator ELSE address generator
CASE...OF (address-generating list)
```

The IF...THEN...ELSE... and CASE...OF functions must contain address-generating functions only if they appear to the left of an assignment or replacement operator.

SECTION 4

STATEMENTS

GENERAL.

Statements are the UPL equivalent of grammatical sentences. They contain a complete sequence of operations (one complete idea) that is logically separate from other similar sequences. While an expression evaluation results in a numerical value, statement evaluation specifies functions or assignments for the values. For example, the expression A+B results in a numerical value; but the statement X := A + B; (read X is replaced by A + B) assigns the value of the expression to data-name X.

Statements are always terminated by a semicolon.

Statements fall into three general classifications: declaration, control, and assignment.

DECLARATION STATEMENTS.

Declaration statements relate memory space to data-names and their attributes. They are described in detail in section 5.

CONTROL STATEMENTS.

Control statements determine the sequence in which statements are to be executed. They pass control to procedures, bind groups of statements together, or conditionally specify which one of several statements is to be executed next.

PROCEDURE CALL STATEMENT.

The major control statement in UPL is the procedure-calling or invoking statement. It consists of a procedure-name followed by any parameters enclosed in parentheses and terminated by a semicolon. For example, the procedure ABS, which requires one parameter, would be invoked by ABS (VALUE); .

There are three considerations governing the use of procedure-calling statements. First, a called procedure must be within the scope of the calling procedure.

In lexic level terms, a called procedure must be:

a. One lower lexic level nested within the calling procedure,

b. Not more than one lower lexic level within a currently invoked procedure that itself is on an equal or higher lexic level, or

c. A currently invoked procedure on an equal or higher lexic level.

Second, a called procedure <u>always</u> returns control back to its calling procedure. There is no GO TO statement in UPL. The programmer must, therefore, structure the program logic to use this return-control action. The immediately succeeding executable statement in the calling procedure is executed when control is returned.

Third, the called procedure must be of the proper class. There are two classes of procedures in UPL. One, which does not pass back a value when it returns control, is referred to as a regular procedure; the other, which does pass back a value, is referred to as a function procedure. The function-procedure call is considered an expression, not a statement. It is described in section 3.

DO STATEMENT.
Statements may be bound or grouped together by the DO, IF...THEN... ELSE, or CASE, statements. The DO statement binds all following statements to its matched END statement as if they were one statement. For example:

```
DO SETA;
  X := X+1;
  A.PARM := ZERO;
  ROUTINE (X, A.PARM);              (This is a procedure call
END SETA;                            with two parameters.)
```

Once a DO group is started it is completed. The individual statements within the group can, however, be any executable statements including imbedded DO statements.

DO FOREVER STATEMENT.
The DO FOREVER statement performs iterations of the statements within the group until an UNDO statement is executed or control is returned from the procedure in which the DO FOREVER is imbedded. For example:

```
DO PRTN FOREVER;
 X := X+1;
 ROUTINE (X, A.PARM);           (procedure call)
 IF X EQL 5 THEN UNDO;          (test limit)
 IF X EQL 10 THEN RETURN;       (return from the current procedure)
END PRTN;
```

IF STATEMENT.
The conditional-expression within the IF statement, when evaluated, designates which of two statements is to be executed. The DO group statement is often used with the IF statement in order to execute a set of statements conditionally. For example:

```
IF A+B GTR X THEN DO;
       A := A-1;
       IF A EQL 0 THEN UNDO;
       RTN.XYZ;
       END;
ELSE DO;
       X := A+B;
       A := 0;
       B := 0;
       END;
```

After the chosen statement executes, control passes beyond the end of the IF statement.

CASE STATEMENT.
The CASE statement is an expanded form of the IF statement. The evaluation of a conditional expression chooses one statement from among all the following statements up to the END CASE statement for execution. After that one statement is executed, control passes to the first statement beyond the END CASE statement.

CASE, DO, IF, and PROCEDURE invocations or assignment statements may be imbedded in any of the above statements in any order and to any depth.

ASSIGNMENT STATEMENT.
The assignment statement is the primary data-movement statement in UPL. Truncation and padding are performed across the assignment operator and are dependent upon the type and length attributes of the data-names as given in the declaration statements.

SECTION 5

DECLARATION STATEMENTS

GENERAL.
Declaration statements specify memory allocation and data attributes or link together portions of a program. They can also be compiler directory information, as in the DEFINE or FORWARD PROCEDURE statement.

This section describes each declaration statement in alphabetical order. The format of each statement is as follows:

a. Purpose.
b. Syntax.
c. Description.
d. Examples.

DECLARE STATEMENT.

The DECLARE statement reserves memory space for data-names and assigns type, length, and hierarchical attributes.

SYNTAX.

The syntactical structure of the DECLARE statement is as follows:

Normal Format

```
DECLARE [PAGED] (page-size) data-name-1 [(array-size)] [REMAPS
    data-name-2] { FIXED
                 { CHARACTER (length) } ;
                 { BIT (length)
```

List Format

```
DECLARE (data-name-1 [(array-size)]
   [[, data-name-2 [(array-size)]] ... ] )
     { FIXED
     { CHARACTER (length) } ;
     { BIT (length)
```

Structured Format

```
DECLARE level-number { data-name-1 [(array-size)] }
                     { FILLER                     }
                     { DUMMY                      }
    [REMAPS data-name-2] { FIXED
                         { CHARACTER (length) } ;
                         { BIT (length)
```

Dynamic Format

```
DECLARE DYNAMIC data-name-1 { CHARACTER (data-name-2) } ;
                            { BIT (data-name-3)       }
```

DESCRIPTION.
DECLARE statements must appear at the beginning of the program or at the beginning of a procedure.

The word DECLARE need not be repeated when commas separate repetitive declarations. Repetitive declarations within one statement can mix format types.

The most efficient code is generated if all data-names are declared in one statement.

Data-name 1 is the name to be assigned memory space and attributes. The data-name must begin with a letter; may contain letters, digits, or dots; must not exceed 63 characters; and may not contain imbedded blanks.

The PAGED option applies to arrays only. It allows the programmer to specify the number of array elements to be contained in one page that is in one segment. The page size must be a power of 2. The maximum number of bits in a page (page size times bit length) is 65,535. The maximum number of characters in a page is 8191. PAGED arrays cannot be part of a structure and cannot be remapped.

Page accessing is performed automatically during program execution.

Array-size specifies the number of elements in an array. Parentheses are required. Arrays are not dimensional and begin at subscript 0 (zero). Array elements are referenced 0 through N-1 for an N element array. The maximum array size is 65,535 elements. Maximum element sizes are 65,535 bits or 8191 characters.

REMAPS re-assigns the memory space of data-name 2 to data-name 1. The type (length) attributes apply to data-name 1, but cannot exceed the length in bits of data-name 2. Any data-names that have been declared can be remapped, including those that remap other data-names.

The type (length) attributes are as follows:

a. FIXED is a sign and a 23-bit binary integer.
b. CHARACTER is an 8-bit unit.
c. BIT is a 1-bit unit.

Detailed specifications regarding data types can be found in section 2.

The (length) specifies how many of the CHARACTER or BIT units to assign. The maximum lengths are as follows: CHARACTERS 8191 and BITS 65,535.

The list format allows repetitive data-names all of the same type (length). They can be of different array-size.

The structured format creates a hierarchy of data-names. The level-numbers assign positions in the structure. The 1 or 01 level must be the first level in any structure. All higher numbers define lower

levels and reDECLARE the memory space of their higher levels. All elements within a structure must contain level numbers. A maximum of 99 levels is allowed for one structure.

Any data-name that is further substructured is called a group data-name. A data-name with no substructure is called an elementary data-name. Elementary data-names must contain a type (length) specification. The group data-names need not contain type (length) specifications. They are assigned the type BIT by the compiler, and their length is the sum in bits of all the elementary data-names below them in the structure.

The word FILLER can be used to avoid naming portions of structures that are not referenced. If the lower portion of any level in a structure does not allocate all of its higher level, the compiler supplies a FILLER. FILLER cannot be used for group data-names.

The word DUMMY is used only with the REMAPS option. It is used to avoid the naming of data-name 1. A DUMMY cannot remap another DUMMY, and it must have at least one non-FILLER data-name.

The maximum number of data elements within one structure, including FILLERs, DUMMYs, and implicit FILLERs, is 64.

Arrays are mapped as continuous areas of memory. If the array is part of a structure, its substructure re-allocates each of the array elements. That is, the substructure array is not mapped into contiguous areas of memory. Each subelement is implicitly an array and must be addressed with subscripts. Structured arrays cannot contain arrays as subelements.

The dynamic format allows simple data-names to be DECLARED with their field lengths for calculation upon each occurrence of the declaration at execution time. The dynamic format can be used only at lexic level 1 or greater. That is, it must be contained within some procedure. Data-name 1 must be either of type BIT or CHARACTER and is assigned a length in units of the binary value of data-name 2 or data-name 3. Data-name 2 or data-name 3 must be DECLAREd, be initialized, and be within scope before the procedure that contains the dynamic declaration is invoked. Data-name 2 or data-name 3 can also be format declarations within the procedure in which they appear in a dynamic declaration. Dynamic declarations cannot contain arrays or structure data-names.

Dynamic declarations can be remapped, but it is the programmer's responsibility to ensure that the remapped declaration does not exceed the length of the dynamic data-name. Unpredictable data results if references are made beyond a dynamic data-name via the REMAPS referencing technique.

NOTE

There is no syntax check made for the initialization of data-name 2 or data-name 3.

The scope of declarations is the same as the scope of the procedure in which they are DECLAREd. That is, they are addressable in their declaring procedure and in any nested procedures. They are not addressable in any global procedures. This addressability is a result of the execution-time, memory-space allocations of declarations.

The execution allocation of memory space is entirely dependent upon the execution sequence of procedures. When a procedure is entered, memory space is allocated for every data declaration in the procedure. When a procedure is exited, that is, a RETURN statement is executed, the declaration memory space becomes available for re-assignment. Data-names declared within a procedure are, therefore, available to the procedure and all of its nested procedures. They are unavailable to any of its global procedures, that is, procedures in which it is nested.

Duplicate data-names can occur in different procedures at compile-time. If the scope of the names overlaps, the compiler references the name at the lower lexic level (within the nested procedure) within the overlapped area. A duplicate data-name within one procedure is a syntax error.

Recursive invocation of a procedure results in a new memory-space allocation for every data-name declared in the procedure.

All data that is used as global data for the entire program must be completely DECLAREd before any other statement types. That is, they must appear in the source program before all nested procedures and before any executable statement.

Class I reserved words cannot be used as data-names.

Class II reserved words can be used as data-names; but, if used, they lose their significance for the scope of the declaration. A warning message appears on the print-out.

Class III reserved words can be used anywhere without confusion except within the specific statement and position that requires them; for example, PUNCH can be declared a data-name. PUNCH in the FILE statement DEVICE portion refers to the I/O device and not the data-name. Class III reserved words may not be defined when used as reserved words.

EXAMPLES.
Examples of the DECLARE statement are as follows:

Examples	Comments
DECLARE TAGA FIXED;	TAGA is a signed 23-bit binary value. The sign is the most-significant (left-most) bit.
DECLARE TAGB CHARACTER (1);	TAGB is of type CHARACTER and 1-unit long. The type

Examples	Comments
	CHARACTER is in 8-bit EBCDIC format.
`DECLARE TAGC BIT (17);`	TAGC is of type BIT and is 17 bits in length.

`DECLARE TAGA FIXED,`
`TAGB CHARACTER (1),`
`TAGC BIT (17);`
	Same as preceding examples except it is a single statement with the items separated by commas.

`DECLARE NAMES (12) CHARACTER (25);`	NAMES is an array of 12 items each with 25 characters per item.

`DECLARE`
`01 CARD CHARACTER (80),`
`02 INPUT CHARACTER (72);`
	A FILLER of eight characters is automatically assigned by the UPL Compiler to round off the 02 level to its required length of 80 characters.

`DECLARE`
`01 TABLE.A CHARACTER (15),`
`02 ITEM.1 CHARACTER (6),`
`02 ITEM.2 CHARACTER (4),`
`03 SUB.ITEM.2 FIXED,`
`/* There is an implicit FILLER of`
`8 bits here */`
	A table of five items that consumes 15 bytes is DECLAREd. Each item is explicitly named in the structure, and its type and length are given. Also DECLAREd is a second table of 200 bits.

`02 ITEM.3 BIT (1),`
`02 ITEM.4 FIXED`
`02 ITEM.5 BIT (7)`
`/* DECLARES may be continued with`
`appropriate commas */`
`01 TABLE.B BIT (200);`
	The SUB.ITEM.2 further subdivides ITEM.2 and uses the first (left-most) three characters (24 bits). The /*...*/ is a comment.

`DECLARE CARDS CHARACTER (80),`
`COLUMNS (80) REMAPS CARDS`
`CHARACTER (1),`
`01 NUM.FIELDS (40) REMAPS`
`CARDS CHARACTER (2),`
`02 FIRST.NUM CHARACTER`
`(1),`
`02 SECOND.NUM CHARACTER`
`(1);`
	An 80-column card is DECLAREd and then remapped as an array of 80 elements, each element of one character. The card is again remapped as a 40-element array, each of two characters. Each 2-character array element is further subdivided into separate elements that can be referenced. Notice that FIRST.NUM and SECOND.NUM must be subscripted when they are used and that 39 is the maxi-

Examples	Comments
	mum value of the subscript.
DECLARE (ITEM.1, ITEM.2, ITEM.3) FIXED;	A list of data-names is DECLAREd, all of type FIXED.
DECLARE 01 NEW.LABEL, 02 NL.1 CHARACTER (25), 02 NL.2 (3) CHARACTER (25), 03 FILLER CHARACTER (5),	A group item NEW.LABEL is DECLAREd, and the compiler assigns it type BIT. It is equal to the sum of the bits of the 02 level below. ((25 + 3 * 25) * 8 + 24 = 824 bits)
03 FIRST CHARACTER (10), 03 SECOND CHARACTER (10), 02 NL.3 FIXED;	NL.2 is an array of three elements each 25 characters in length. FILLER is used to omit the naming of an area that is never referenced separately. FILLER can be used as often as required without causing a duplicate-name syntax error. FIRST and SECOND are 3-element subarrays of the NL.2 array. They are referenced with subscripts 0, 1, and 2, for the first, second, and third elements. Each element is 10 characters. NL.3 is a FIXED signed binary number.
DECLARE 01 A, 02 A1 (20) BIT (20), 02 A2 (18) BIT (20), 03 B1 BIT (15), 03 B2 BIT (5), 02 A3 (2) BIT (5);	The data-names A1, A2, B1, B2, and A3 must all be subscripted, when used, because of the explicitly declared array-sizes specified for A1, A2, and A3. The length sum of data-names B1 + B2 must be equal to, or less than, that specified for data-name A2.
DECLARE 01 TAGA (5) BIT (48), 02 TAGB FIXED, 02 TAGC FIXED;	TAGA is mapped in a contiguous memory area to contain the data developed for TAGB and TAGC. TAGB and TAGC are implicit 5-unit arrays, but are <u>not</u> mapped contiguously. They are mapped alternatingly as follows: TAGB(0), TAGC(0),

Examples	Comments
	TAGB(1), TAGC(1),. . . ., TAGB(4), TAGC(4).
DECLARE PAGED (64) BIG.D.N. (5000) BIT (1);	BIG.D.N. is an array of 5000 elements, each of one bit. The array is segmented into 64 parts. Each part is brought into memory, that is, paged, whenever it is addressed. No special statements are required to do the paging.

DEFINE STATEMENT.

The DEFINE statement provides the capability of inserting multiple copies of specified UPL source text from only one image of the source text into a program during compilation.

SYNTAX.
The syntactical structure of the DEFINE statement is as follows:

```
DEFINE  definition-name  [ (parameter-1  [, parameter-2]... ) ]

     AS # text [parameter-1    [text   parameter-2]  ...] # ;
```

DESCRIPTION.
The data specified between the # signs, called the text, is paired with the definition-name. When the definition-name appears in any subsequent location in the program, the compiler replaces the definition-name with the text. The text must conform to the syntactical requirements of the statements into which it is placed.

The DEFINE statement must appear within the declaration section of the program or of a procedure. The scope of a DEFINE statement is the same as the scope of any data-names in that declaration section. That is, the scope exists in its declaring procedure and all directly nested procedures. Multiple DEFINEs can appear within one DEFINE statement and must be separated by commas. DEFINE statements can be nested to a depth of 12 levels. That is, the text can contain previously declared definition-names. The compiler expands all nested DEFINE statements into their appropriate text strings.

The text can contain any UPL symbol including semicolons, but it cannot contain the # or % signs. The # sign is the text delimiter, and the % sign indicates that the remainder of the card is a comment. The comments sign, /*...*/, can appear within a DEFINE statement, but it is not copied at invocation time. A maximum of 1024 characters can appear in a DEFINE string, excluding comments and superfluous blanks. Also, no unpaired bracketing symbols (() or []) may appear within a define.

All data-names that are coded within the text must be DECLAREd prior to an invocation of the definition-name, but need not be DECLAREd prior to the DEFINE statement.

Re-usage, that is, duplication, of a definition-name on a lower lexic level for any of the following names inhibits the substitution of the text for the scope of the duplicate name. The names are as follows:

a. DECLARE names.
b. PROCEDURE names.
c. FORMAL names.
d. SEGMENT names.
e. DO group names.
f. FILE, OPEN, WRITE, and CLOSE attributes.

A duplicate name within scope and on the same lexic level is a syntax error.

Duplicate definition-names, which can be encountered between lexicographical levels, are resolved at compile-time by using the most current name (on the highest level). When a lexicographical level is exited (exit from a procedure), the names on the higher lexicographical level are lost and can be re-used. Reserved words of class I cannot be used as definition-names. A definition-name can, however, define a reserved word. Reserved words of class II can be used as definition-names, but their special significance is lost within the scope of the DEFINE statement.

The actual parameters associated with an occurrence of a definition-name are not restricted to simple data-names. They can contain complex constructs, but must be delimited by 0-level commas, that is, commas not enclosed within paired parentheses or braces.

The actual parameters replace the format parameters in the DEFINE statement in a left-to-right order, and their number must be equal. The maximum number of parameters is limited to eight per definition-name.

Conditional compile cards (cards with an & sign in column 1) may appear as part of a DEFINE string. A $DETAIL compiler option card prints the expansion of DEFINEs on the compiler print-out.

EXAMPLES.
Examples of the DEFINE statement are as follows:

Examples	Comments
`DEFINE REPEAT AS #ABC (TAGA, X) #;`	The source code contained between the # signs of the DEFINE statement is copied into the UPL Program whenever the word REPEAT is used.
`IF X EQL 9 THEN REPEAT;`	This statement is equivalent to IF X EQL 9 THEN ABC (TAGA, X);

`DEFINE CH AS # CHARACTER #,`
`  FX AS # FIXED #`
`  THEN DECLARE X CH (5),`
`  Y FX, Z CH(2);`
	The source code generated would be:
	`DECLARE X CHARACTER (5),`
	`  Y FIXED,`
	`  Z CHARACTER (2);`

`DEFINE TRIAL (A,B,C) AS`
`  # IF (A) EQL ZERO THEN A := B;`
`  ELSE C #;`
	This statement generates the following: IF (TAGA) EQL ZERO THEN TAGA := ABS

Examples	Comments
	(BX); ELSE CX := SQRF (BX);
DEFINE T AS #@(1)1@, F AS #@(1)0@;	The T and F become Boolean bit strings of 1 or 0, respectively.
DEFINE X AS # ABC # ABC AS # X #;	This statement causes an error diagnostic at compile-time when the compiler attempts to expand either X or ABC into TEXT.
DEFINE MAX AS # & IF S1 A := X; & ELSE A := Y; & END #;	This whole statement is available to the compiler, but only A := X or A := Y is compiled, depending on the condition of the conditional symbol S1. IF the statement & SET S1 has been encountered, A := X; is used. IF S1 has not been set or is reset, that is, & RESET S1, then A := Y is used.

DEFINE A AS #IF X GTR 10 THEN
PROCX#,
C(M) AS #X := M; A #;
C(Z; BUMP I (R + S))
	This statement expands to X := Z; BUMP I (R+S); IF X GTR 10 THEN PROCX;

DEFINE MAX.SIZE AS & IF DATACOMM 64 & ELSE 32 & END	If a conditional compile card of & SET DATACOMM appears, the DEFINE MAX.SIZE would result in 64. If the &SET does not occur or if a &RESET DATACOMM occurs, MAX.SIZE is defined as 32.

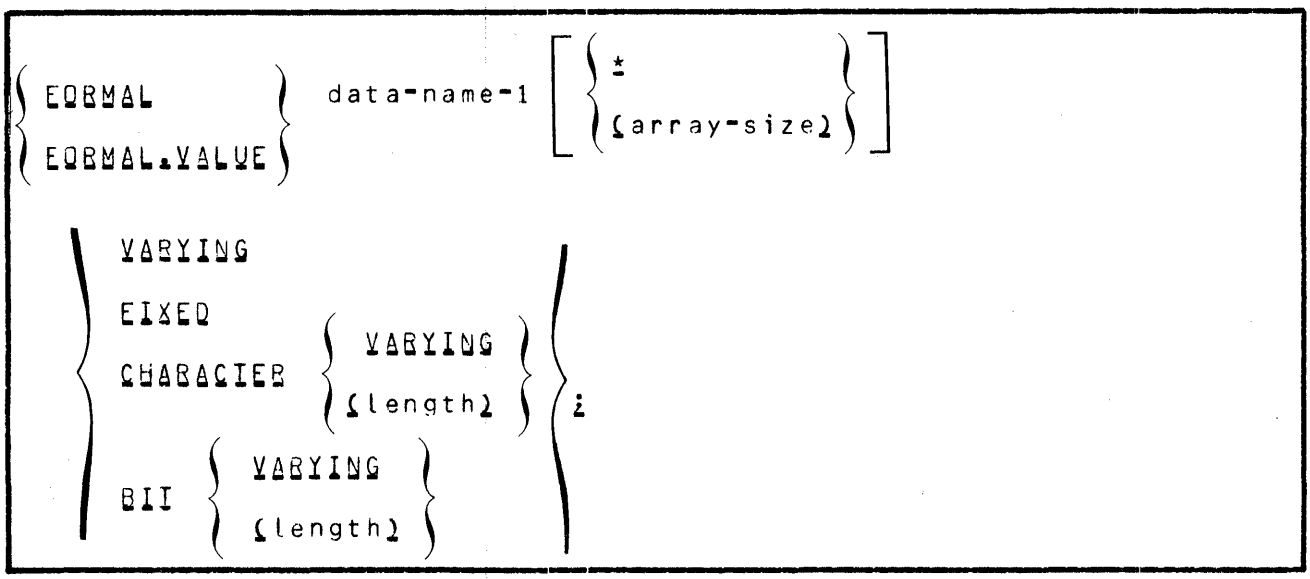
FORMAL STATEMENT.

The FORMAL statement is used to assign data attributes to the parameter-name in the procedure head statement and the FORWARD PROCEDURE statement.

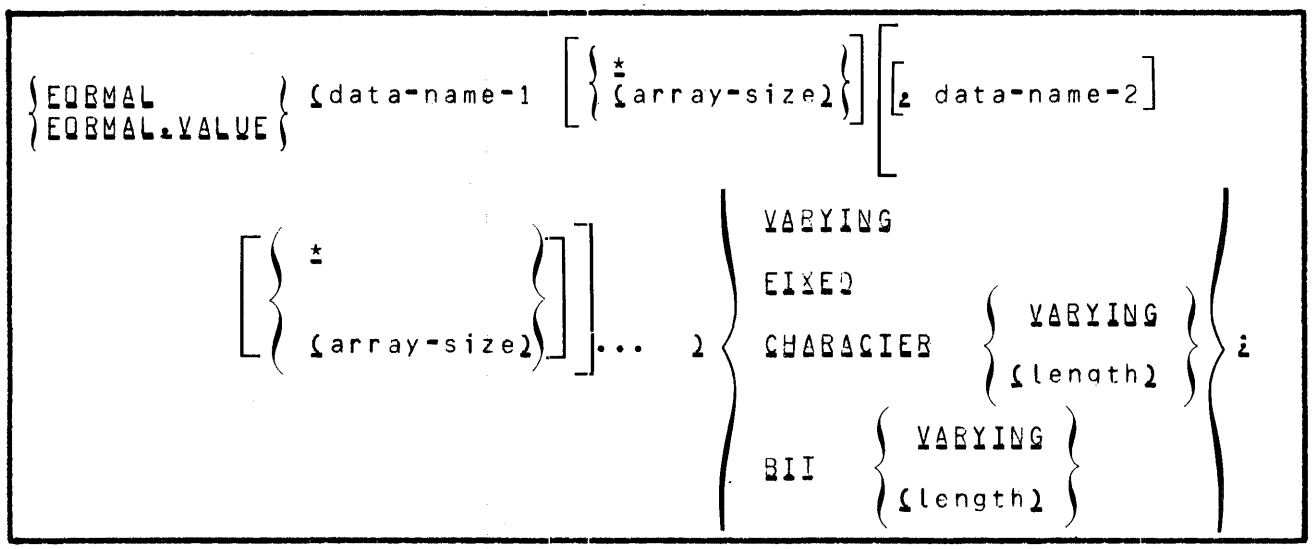
SYNTAX.

The syntactical structure of the FORMAL statement is as follows:

Normal Format



List Format



DESCRIPTION.

All of the data-names in the procedure head statement and FORWARD PROCEDURE statement must be declared in the FORMAL statement. Only the data-names in the procedure head statement or a FORWARD PROCEDURE statement may appear in a FORMAL statement.

The data-names in the procedure head or FORWARD PROCEDURE statements should agree in type and length attributes with the actual data-names that are passed at object-time. Run-time checking is performed only if the compiler option $FORMAL CHECK is requested. No checking cccurs during compilation. If checking is requested and a mismatch cccurs, the program is terminated.

The names given in a FORWARD PROCEDURE statement, however, need not agree with their corresponding names in the procedure head statement. The type and length attributes of the data-names in the FORWARD PROCEDURE statement and the procedure head statement must agree.

Object-time adjustment of type and length attributes between the actual parameters passed and the specified formal parameter data-name is performed with the VARYING option or array-size with the * (asterisk) option. Memory allocation is dependent on the actual parameters passed during each invocation of the procedure.

The words FORMAL or FORMAL.VALUE can be omitted between repetitive declarations when the declarations are separated by commas.

The type (length) attribute specification in the list format applies to all the data-names in the immediately preceding list.

The FORMAL statement must appear immediately after the procedure head statement or the FORWARD PROCEDURE statement. That is, the FORMAL statement must appear before any local data declarations within a procedure.

Level numbers are not allowed in a FORMAL declaration.

Data-names that appear in a FORMAL statement can be remapped by local data declarations. If they contain the VARYING or * options, a warning message appears on the print-out. It is the programmer's responsibility to ensure proper remapping.

The FORMAL.VALUE option causes the actual parameter always to be passed-by-value.

EXAMPLES.
Examples of the FORMAL statement are as follows:

Example	Comment
PROCEDURE ABC (X, Y, Z); FORMAL X FIXED, Y CHARACTER VARYING, Z (*) BIT VARYING;	Procedure ABC has three parameters that must be declared FORMALly. X is a simple FIXED data-name. Y is of type CHARACTER. The length is calculated on each call of the procedure. Z is an array of a varying number of elements of type

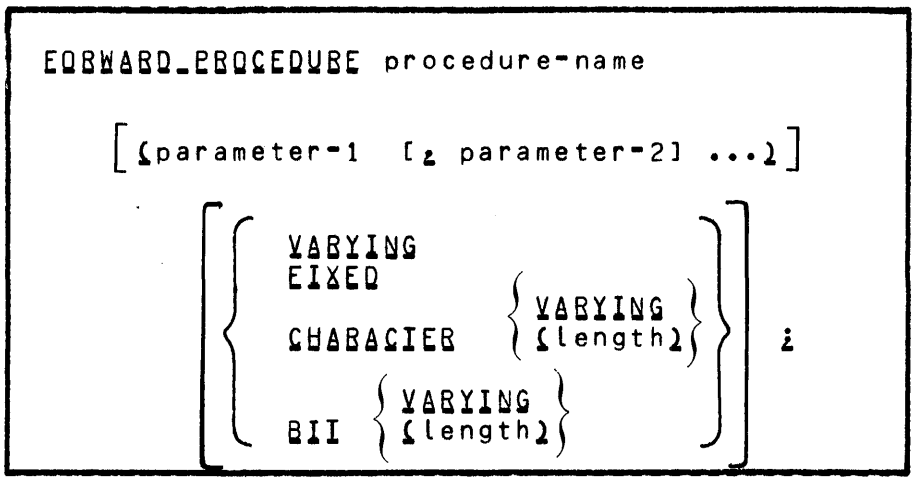
Example	Comment
	BIT where each element is also calculated on each call of the procedure.

FORWARD PROCEDURE STATEMENT.

The FORWARD PROCEDURE statement declares a procedure-name as being valid prior to the encountering by the compiler of the named procedure.

SYNTAX.

The syntactical structure of the FORWARD PROCEDURE statement is as follows:



DESCRIPTION.

The FORWARD PROCEDURE statement is the same as, and must conform to, the rules of the procedure head statement. It is, however, a declaration statement and, as such, must appear within a DECLARE section of a procedure or of the program.

The FORWARD PROCEDURE statement is a compiler control-statement, and its presence does not eliminate the syntactical requirement of a procedure head statement. It resolves all forward address references for a call to a procedure when the procedure has NOT yet been seen by the compiler. This is necessary because UPL is a 1-pass compiler.

The FORWARD PROCEDURE statement must contain its named procedure within scope. That is, the FORWARD PROCEDURE statement must be global to the referenced procedure. The type (length) clause is used only with function procedures. It specifies the type and length of the value that is returned when the function procedure passes back control. Use of the VARYING option inhibits the checking of length or type and length of the returned value.

When a FORWARD PROCEDURE statement contains a parameter, a FORMAL declaration statement must immediately follow with the same data-names as those in the FORWARD PROCEDURE statement. The types and lengths of the parameters that are used in the FORMAL statement must correspond with the types and lengths that are declared FORMALly in the procedure. The data-names, however, need not be the same.

EXAMPLES.
Examples of the FORWARD PROCEDURE statement are as follows:

Examples	Comments
FORWARD PROCEDURE X;	Procedure X is being FORWARD declared. It may be referenced after this statement and before the procedure is actually encountered by the compiler.
FORWARD PROCEDURE ABS (H, I, J) BIT VARYING; FORMAL (H, I) FIXED, J, CHARACTER (4);	Procedure ABS has three parameters that are also declared FORMALly. The procedure is a function procedure that returns a VARYING length bit-string.

PROCEDURE STATEMENT.

The PROCEDURE statement is used to delineate a group of statements and their data declarations. Also, it provides a method whereby the group can be executed from many places although the procedure appears only once.

SYNTAX.

The syntactical structure of the PROCEDURE statement is as follows:

```
PROCEDURE name-1 [(parameter-1 [, parameter-2 ...] )]
                                [{ FIXED
                                 { VARYING
                                 { CHARACTER { (length) } ;
                                 {           { VARYING }
                                 { BIT { (length) }
                                 {     { VARYING }
[NORMAL ...;]
[DECLARE ...;]
[FORWARD PROCEDURE ...;]
[       [PROCEDURE name-2 ...;]
        [NORMAL ...;]
        [DECLARE ...;]
        [FORWARD ...;]
                        [  PROCEDURE name-n ...;
                           ...;
                           ...;   (Include executable
                                   statements of procedure-n.)
                           ...;
                           END name-n;                    ]
        (Include executable statements for PROCEDURE 2 at this
         point.)
        END name-2;                                       ]
        (Include executable statement for PROCEDURE 1 at this
         point.)
    END name-1;
```

DESCRIPTION.
There are two classes of procedures that differ principally in manner of invocation and in what they pass back to their invoking statements.

a. Regular procedures execute a set of code and RETURN control back to the statement that follows their calling statement.

b. Function procedures execute a set of code and RETURN control and a value into their calling statement. The RETURNed value is, then, used instead of the function designator in the calling statement. The statement evaluation is then continued.

Procedures can be created by surrounding a body of self-contained statements with a procedure head statement and an END statement. The body is divided into two parts: first, the declaration portion in which all local data-names and all nested procedures are DECLAREd; second, the executable portion containing all of the executable statements of the procedure.

The procedure head statement consists of the following:

a. A name that is used to cause an invocation of the procedure.

b. A set of data-names that must agree in number with the parameters actually passed.

c. A type (length) specification if the procedure is to RETURN a value. (Only a function procedure RETURNs a value.)

d. A FORMAL data declaration to specify the type (length) and array size of all parameters named in the procedure head statement. These specifications should, therefore, also agree with those of the actual parameters that are being passed.

At execution time, when a PROCEDURE statement is invoked, the declaration portion performs several functions. Formal declarations assign attributes for all parameters specified in the procedure head statement. Memory space assignment is made for all parameters passed-by-value, and the current value is loaded. Parameters passed-by-name use the original memory space, but are referenced with the name in the FORMAL statement. Declared data-names are assigned memory space and attribute characteristics. Initial values are not loaded and must be supplied by the programmer with executable statements. The possibility exists that the procedure being invoked contains a data-name that is the same as one that is contained in an already invoked procedure. For example, procedure SQU can contain data-name X, and procedure ABS that is being called can also contain data-name X. The duplicate names problem is resolved across procedure bounds by making the most current occurrence of the duplicate data-name available, that is, data-name X in procedure ABS. Data-name X in procedure SQU becomes available again when procedure ABS executes a RETURN, that is, is exited. If no such

duplicate data-name problem occurs, all the data-names of each outer procedure, within any nest of procedures, are available to all nested procedures. For example, if procedure SQU contains data-name N, and procedure ABS does not contain a similar data-name, and if procedure ABS is nested within SQU, then any occurrence of data-name N in either procedure refers to the same memory space.

The FORWARD PROCEDURE statement resolves the forward reference problem for the UPL single-pass compiler.

Nested procedures are part of the declaration portion of a procedure, but must appear after all other types of declarations. They must be completely defined before other nested procedures on the same lexic level and before any executable statements in an outer procedure. That is, procedures must not overlap.

NOTE
A new lexic level is created by nesting a procedure. The limit is 15 lexic levels.

Executable statements are the operations that are performed when a PROCEDURE statement is invoked. Any executable UPL statement may be coded, including procedure-calls and function procedure designators.

The scope of the procedure-name defines the range over which a PROCEDURE can be invoked; therefore, a PROCEDURE can call itself (recursion). The limit for recursion is program available memory for data declarations. Beyond the limit, the MCP aborts the program and generates an error message. Procedure object code is maintained outside the base and limit registers of the program and is re-entrant.

An END statement must contain the name of the PROCEDURE that it ends. A syntax check is performed to guarantee that the END statement is placed properly.

Procedures can contain code segments, but the procedure itself must begin and end within the same segment.

The END statement of the procedure, if executed, is equivalent to a RETURN statement. If the procedure is a function procedure, that is, a value is to be passed back into the invoking expression, the following table shows what types and values are passed if the END statement return is executed.

Procedure Type-Length	Value-Length
BIT (length)	Zero bits of the specified length
CHARACTER (length)	Blanks (hexadecimal 40) of the specified length
FIXED	Fixed 0 (zero)

Procedure_Type-Length	Value-Length
BIT VARYING	Eight bits of 0 (zero)
CHARACTER VARYING	One blank (hexadecimal 40)
VARYING	Fixed 0 (zero)

EXAMPLES.
Examples of the PROCEDURE statement are as follows:

Example	Comment
PROCEDURE SQUARE (N); FORMAL N FIXED; . . RETURN; . . . END SQUARE;	Procedure-name SQUARE is called from some point in a program. A value for data-name (N) is passed by the calling statement.
PROCEDURE CUBE (A, B, C); FORMAL (A, B, C) FIXED; PROCEDURE SQUARE (N); FORMAL N FIXED; . . . IF A THEN RETURN; . . . END SQUARE; . . . IF B THEN RETURN; ELSE DO; SQUARE (C); RETURN; END; END CUBE;	Two procedures, one nested within the other, are declared. The procedure SQUARE can be invoked only from within the procedure CUBE.
PROCEDURE ABSVAL (X) FIXED; FORMAL X FIXED; RETURN (IF X LSS 0 THEN - X ELSE + X); END ABSVAL;	A function procedure returns the absolute value of the actual parameter passed. The IF expression within the RETURN statement passes back the posi-

Example

```
PROCEDURE MSG CHARACTER (20);
    DECLARE DATA CHARACTER (20);
    RETURN (ACCEPT DATA);
    END MSG;
```

Comment

tive value of the parameter.

A function procedure accepts a message from the console printer and passes it back to its invoking statement. For example,

```
IF SUBSTR(MSG, 0, 3) EQL
"YES"
          THEN...;
          ELSE...;
```

SEGMENT STATEMENT.

The SEGMENT statement divides program object code into overlayable sections in order to reduce the run-time memory requirements of the program.

SYNTAX.
The syntactical structure of the SEGMENT statement is as follows:

```
SEGMENT [segment-name] ;
```

DESCRIPTION.
If no SEGMENT statements appear, the whole program is one resident segment. All segments are overlayable.

Segment names must begin with a letter and cannot contain more than 63 characters. If a program is to be SEGMENTed, the first statement within the program should be a SEGMENT statement. If the first SEGMENT statement appears within the body of the program, everything up to and including this segment is the first segment. A warning message appears in the print-out.

Segments may themselves be grouped into pages (refer to SEGMENT.PAGE statement, page 5-25).

When unique segment-names are specified, they imply that unique program segments are to be created at compile-time from the point of insertion. Non-unique names imply that a continuation of an already existing segment is to be continued and is to be gathered by the compiler.

Procedures and DO groups can extend into more than one segment, but must begin and end within the same segment. In general, for efficient programming, procedure and DO groups should be completely contained within the range of a single segment-name.

Certain statements in UPL contain subordinate statements. A SEGMENT statement that immediately precedes a subordinate statement applies only to the subordinate statement. The IF...THEN...ELSE, CASE, READ, WRITE, and SPACE statements contain subordinate statements.

At run-time, no UPL statements are required to access a non-resident program segment. Such action is entirely the responsibility of the MCP.

NOTE
Array data-names can also be memory partitioned. Refer to the PAGED option in the DECLARE statement.

EXAMPLES.
Examples describing the use of the SEGMENT statement are as follows:

Example	Comment
SEGMENT (ONE);	First segment.
... ... (statements) ...	
SEGMENT (TWO);	Second segment.
... ... (statements) ...	
SEGMENT (THREE);	Third segment.
... ... (statements) ...	
SEGMENT (TWO);	This segment is gathered with the second segment.
... ... (statements) ...	
SEGMENT (FOUR);	Fourth segment.
... ... (statements) ...	
SEGMENT (TWO);	This segment is gathered with the second segment.
... ... (statements) ...	
SEGMENT (FOUR);	This segment is gathered with the fourth segment.
... ... (statements) ...	
SEGMENT (N);	

In the above example, program control is not affected by the gathering technique. The advantages of using such a technique allow for optimum use of memory allotment at run-time.

IF TEST EQL OK THAN ABX := 4; ELSE SEGMENT (ERROR); DO; MARK.ERR; RETURN; END;	The statement following ELSE is in a separate segment and is called in when the data-name TEST does not equal data-name OK. MARK.ERR is a procedure call and should also be in the segment ERROR.

SEGMENT.PAGE STATEMENT

The SEGMENT.PAGE statement allows program code dictionaries and the corresponding code segments to be paged, thus reducing dictionary memory requirements for programs with many segments.

SYNTAX.
The syntactical structure of the SEGMENT.PAGE statement is as follows:

```
SEGMENT.PAGE ( segment-name OF page-name ) ;
```

DESCRIPTION.
The segment-name specifies the name of a code segment to the compiler. Non-unique names within a page are gathered. Segment-names must be unique regardless of which pages contain them.

The SEGMENT statement can also occur in the source language, and the effect depends upon the uniqueness of the segment-name.

Unique segment-names that follow a SEGMENT.PAGE statement are contained in that page.

Two classes of non-unique segment-names are possible. One is the re-occurrence of a segment-name in the current page. The segments are gathered within the page. The second is the re-occurrence of a segment-name not in the current page. The current page is altered and the segment is gathered properly. A following unique segment-name is included with the new current page. A warning message also appears on the print-out when the page is implicitly altered.

Sixty-four unique names are allowed in one page.

The page-name specifies the name of the segment-dictionary. Non-unique names are a continuation of the existing dictionary. Sixteen unique page-names are allowed.

EXAMPLES.
Examples describing the use of the SEGMENT.PAGE statement are as follows:

Example	Comment
SEGMENT.PAGE (AA of ONE);	A paged segment dictionary is created with one entry called ONE.
SEGMENT (BB);	Page ONE contains segments AA and BB while page TWO contains segments CC and DD. The re-occurrence of the SEGMENT (BB) statement changes the current page back to page ONE. The SEGMENT.PAGE (DD

Example	Comment
SEGMENT.PAGE (CC of TWO); • • • • SEGMENT (DD); • • • • SEGMENT (BB); • • • • SEGMENT.PAGE (DD OF TWO);	OF TWO) is therefore required to continue the DD segment.

USE DECLARATION STATEMENT.

The USE statement declares specific data-names in a defined structure within a procedure.

SYNTAX.
The syntactical structure of the USE statement is as follows:

```
USE ( data-name-1[[, data-name-2]...])[ OF data-name-3];
```

DESCRIPTION.
The USE statement allows the programmer to declare only those data-names desired within a structured list of data-names. The compiler generates FILLER wherever necessary to complete the structure. This results in more rapid procedure entrance and less memory utilization than if the whole structure were declared upon each entrance to the procedure.

The USE statement must appear within a procedure. That is, it may not be used on lexic level 0.

The structure being referenced must have as its 01 level a DUMMY REMAPS declare, must not contain arrays, and must be contained within a DEFINE statement.

The DEFINE statement may contain only the one structure.

EXAMPLES.
Examples describing the use of the USE statement are as follows:

Example	Comment
DECLARE PPB BIT (1440);	The space to be remapped.
DEFINE PPB. DEC AS #	The DEFINE for the USE statement.
DECLARE 01 DUMMY REMAPS PPB,	The required DUMMY 01 level.
02 PROG.NAME CHARACTER (10),	Remaps and the layout of memory spare.
02 PROG.DATA.DICT BIT (112),	
02 PROG.SEC.DICT BIT (112),	
02 PROG.SORT.SPAD BIT (28) #;	

Example	Comment
PROCEDURE GET.DICT;	The PROCEDURE in which the USE statement appears.
USE (PROG.DATA.DICT, PROG.SEC,DICT) OF PPB.DEC;	Only two memory spaces are allocated as addressable. The rest of the data-name PPB is considered FILLER.

SECTION 6

EXECUTABLE STATEMENTS

GENERAL.
Executable statements perform the data transformations and the decision-making functions of a UPL Program.

Executable statements are given in alphabetical order. The format of each statement is described in the following order:

a. Purpose.
b. Syntax.
c. Description.
d. Example.

ARRAY_PAGE_TYPE_STATEMENT.

The array page type statement specifies whether a page need be written to disk when it is no longer needed in memory.

SYNTAX.
The array page type statement syntax is as follows:

```
{ MAKE.READ.ONLY  }  ( paged-array-name, {page-number  }  )
{ MAKE.READ.WRITE }                      {expression   }
```

DESCRIPTION.
All paged arrays are originally read/write. A page can be made read-only after it has been initialized. It is then not written to disk each time, and it is no longer required in memory. It can be made read/write again with the MAKE.READ.WRITE statement.

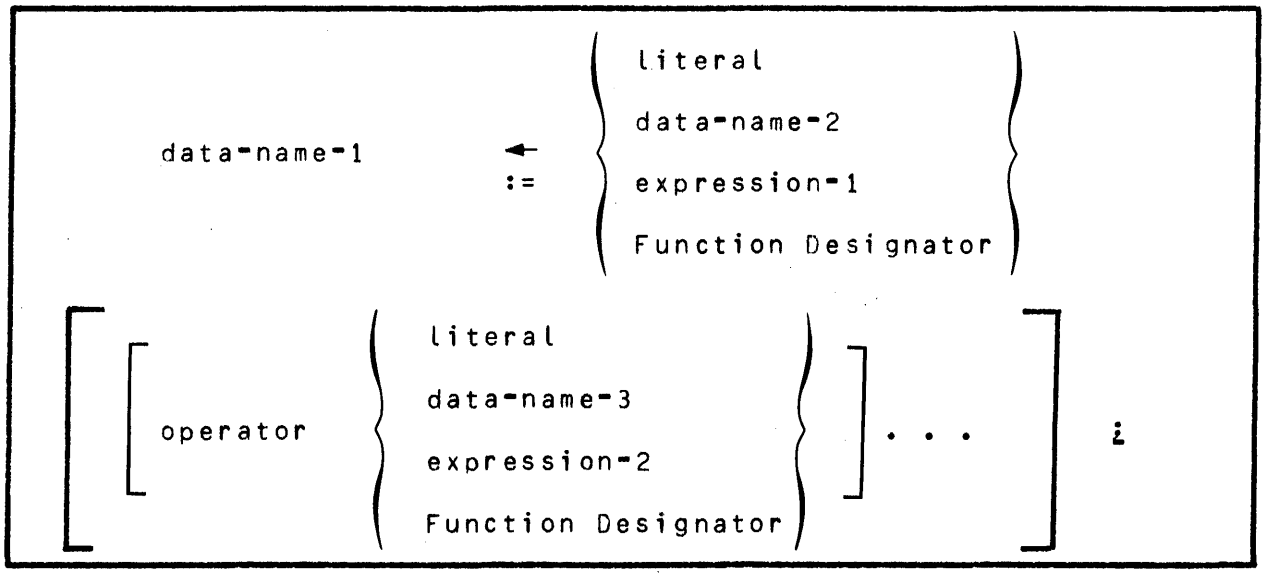
Paged-array-name must be an array that has been declared PAGED.

Page-number-expression must result in a valid number from 0 to N-1 for an N page array. It is the programmer's responsibility to calculate the number of each page being specified in the array page type statement.

ASSIGNMENT STATEMENT.

The assignment statement is used to assign the value of an expression to a specified data-name.

SYNTAX.
The syntactical structure of the assignment statement is as follows:



DESCRIPTION.
The characters := (colon, equal) and ← (left arrow) are called the assignment symbols and are read "is replaced by."

The value of the total expression to the right of the assignment symbol is assigned to data-name 1.

A data-name must have been previously declared before it can be used. Each data-name may be of different type (length) across the assignment symbol with justification and alignment as follows:

Declaration Types	Comments
FIXED to FIXED	No change occurs.
CHARACTER to CHARACTER	Data is left-justified with least-significant truncation or space (hexadecimal 40) fill.
All others	Data is right-justified with most-significant truncation or zero fill.

The optional operator clause implies one of the following entries, which are in order of precedence:

Operator	Comments
:= or ←	Replace and delete left part.
::= or :←	Replace and delete right part. Must be higher than any operator to its right and lower than any operator to its left.
+	Unary plus.
-	Unary minus.
*	Multiplication.
/	Division.
+	Addition.
-	Subtraction.
LSS or <	Less than.
LEQ or ≤	Less than or equal to.
EQL or =	Equal to.
GTR or >	Greater than.
GEQ or ≥	Greater than or equal to.
NEQ or ≠	Not equal to.
NOT	Explicit NOT logic.
AND	Explicit AND logic.
EXOR	Explicit EXclusive OR logic.
OR	Explicit OR logic.
CAT	Concatenate.
MOD	Provides the remainder of division.
:= or ←	Replace delete left part.
::= or :←	Replace delete right part. Must be higher than any operator to its right and lower than any operator to its left.

A data-name or an address-generating function (refer to SUBSTR, page 8-25) must appear to the left of replacement operators within an expression in the assignment statement. A data-name, however, may be the object of another replacement operator in the expression, such as A := B + X := C; or A(I) := I := B;. Any assigned data-name is altered only from its points of assignment in a left-to-right order. Any previous reference to the data-name to the left within the overall statement retains its prior value, and any other reference to the right uses the newly assigned value.

The replacement operator (::= or :←) is similar in function to the usual assignment symbol, := or ←. The two differences are: first, that the replacement operator must be used within an expression, and, second, that the address of the data-name to be used with the next term during expression evaluation is the data-name to the left of the operator.

The unary operators (+, -, NOT) may be used to the right of any other operator.

A semicolon terminates the assignment statement.

EXAMPLES.
Examples describing the use of the assignment statement are as follows:

Examples	Comments
TAGA := TAGB;	Data-name TAGA is replaced by the value contained in data-name TAGB.
TAGA := @(4)C1@;	TAGA is replaced by a hexadecimal bit string. The bit string is an EBCDIC A.
TAGA := A + B - C * E / F + (4 * (A - B) / (B - C));	Data-name TAGA is replaced by the derived value of the given expression at object run-time.
TAGA := TAGB := TAGC := 0;	The entire set of data-names is set to 0 (zero) at object run-time.

In the following example DECLARE A1 FIXED, B1 CHARACTER (2), C1 BIT (4):

Examples	Comments
Step 1. A1 := B1 := C1 := 0;	All data-names are set to 0.

Examples	Comments
Step 2. A1 := B1 := C1 := 7;	All data-names now contain a value of binary 7 (00...0111).
Step 3. A1 := B1 := "7";	The "7" entry denotes an 8-bit (byte) representation of the value of a numeric 7; therefore, B1 contains F740, or 63,296 in binary.
	NOTE There is no intrinsic conversion between data types. The CONV function must be used if conversion is desired.
Step 4. A1 := 255; B1 := "AA";	A1 now contains the binary value C0...11111111, and B1 contains the hexadecimal equivalent of the characters AA (C1 C1), binary 1100000111000001.
Step 5. B1 := A1;	B1 now contains the hexadecimal value of 00FF, binary 00...00011111111.
Step 6. C1 := B1;	C1 now contains the hexadecimal value of F, binary 1111.
Step 7. B1 := C1;	B1 now contains the hexadecimal value of 000F, binary 0000000000001111.
Step 8. B1 := C1 CAT C1 CAT C1 CAT C1;	B1 now contains the hexadecimal value of FFFF, binary 1111111111111111.

In the following example DECLARE AA (10) FIXED, BB FIXED:

NOTE
AA (10) is a 10-element array.

Examples	Comments
Step 1. `BB := 5;`	Set BB equal to 5 (00...101).
Step 2. `AA (BB) := BB := 3;`	Save the address of AA (BB) or AA(5). Set BB = 3. Then use the address AA(5) and set it equal to 3.
NOTE The above two statements are equivalent to steps 3, 4, and 5.	
Step 3. `BB := 5;`	Set data-name BB equal to 5, then,
Step 4. `AA (BB) := 3;`	Set data-name AA(5) equal to 3, then,
Step 5. `BB := 3;`	Set data-name BB equal to 3.
Step 6. `A1 := BB LSS CC;`	If the condition where BB is less than CC is TRUE, then A1 is assigned 1; if the condition is FALSE, then A1 is assigned 0.

Step 7.
`A1 := (BB LSS CC) +`
`(BB GTR CC);`
	If the content of data-name BB equals the content of data-name CC, then A1 is assigned 0; otherwise, A1 is assigned 1.

Step 8.
`DECLARE (A1, BB, CC, DD) FIXED;`
`BB := 5;`
`CC := 6;`
`DD := 7;`
	These three could have been written as follows:
	`DD := 1 + CC := 1 + BB := 5;`

Step 9. `A1 := BB := CC + DD;`	A1 and BB are both equal to 13.
Step 10. `A1 := (BB := CC) + DD;`	A1 equals 13 and BB equals 6.

The following two examples use the delete left replacement operator.

Examples	Comments
DECLARE (AA,CC) CHARACTER (2), BB BIT (4); AA := BB := CC := "6";	CHARACTER data-name CC is replaced by the value of the CHARACTER literal 6. Data-name CC contains hexadecimal F640; and the CHARACTER data-name CC is deleted (left part), and data-name BB is replaced by the value of the CHARACTER literal 6. Data-name BB contains hexadecimal 6. The BIT data-name BB is deleted (left part), and the CHARACTER data-name AA is assigned the value of the literal 6. Data-name AA contains hexadecimal F640.
X := AA + BB := 6;	Data-name BB is replaced by the literal 6. Data-name BB is deleted, and the literal 6 is added to data-name AA. The sum is then assigned to data-name X.

The following four examples use the delete right part replacement operator.

Examples	Comments
AA := BB ::= CC ::= "6";	The CHARACTER data-name CC is replaced by the value of the CHARACTER literal 6. Data-name CC contains hexadecimal F640. The CHARACTER literal is deleted (right part), and BIT data-name BB is replaced by the value of CHARACTER data-name CC. Data-name BB contains hexadecimal 0. The replacement is type CHARACTER to BIT. Data-name CC is deleted (right part), and CHARACTER data-name AA is assigned the value of data-

Examples	Comments
	name BB. Data-name AA contains hexadecimal 0000.
`X := AA + BB ::= 6;`	Data-name BB is replaced by the literal 6. The literal 6 is deleted, and data-name BB is added to data-name AA. The sum is assigned to data-name X.

`PROCEDURE SQRF (X) FIXED;`
`  FORMAL X BIT (4);`
`  AA := SQRF (BB ::= CC);`
	The delete-right-part is used in the procedure call to force the type and length of the parameter to agree with the type and length in the FORMAL statement of the PROCEDURE.

`X := (AA + BB) := 6;`	This is a syntax error because (AA + BB) is not a data-name or an address-generating expression. It is a value-generating expression.

The following eight examples describe the event order in assignment statements.

Examples	Comments
`DECLARE (A, B, C) FIXED;` `  A := B := C := 0;`	A, B, and C are all equal to 0. The order of events is as follows: C is replaced by 0, C is deleted, B is replaced by 0, B is deleted, A is assigned 0, and the statement is completed.
`A := 1 + B := 1 + C := 0;`	A equals 2, B equals 1, and C equals 0. The order of events is: C is replaced by 0, C is deleted, 1 is added to 0, and B is replaced by the sum. Then, B is deleted, 1 is added to the sum, A is assigned the result, and the statement is completed.
`A := B + 1_:=_C + 1_:=_0;`	The underlined parts are syn-

Examples	Comments
	tax errors. A literal may not be the object of a replacement operator.
A := B ::= C := 0;	The order of events is as follows: C is replaced by 0, C is deleted, B is replaced by 0, 0 is deleted (this is a replace delete right part operator), A is assigned the result. A, B, and C all equal 0.
A_:=_B := C := 0;	Syntax error. The assignment operator must be a delete-left-part.
A := 1 + B := C ::= 0;	The order of events is as follows: C is replaced by 0, 0 is deleted, B is replaced by C, B is deleted, 1 is added to C, and the sum is assigned to A. A now equals 1, and B equals C, which equals 0.
A := 1 + B ::= 1 + C := 0;	The order of events is as follows: C is replaced by 0, C is deleted, 1 is added to 0, B is replaced by this sum, the sum is deleted, 1 is added to B, and this sum is assigned to A. A = 2, B = 1, C = 0.
A := B + 1_:=_ 1 + C := 0;	Syntax error. A literal may not be the object of a replacement operator.

BUMP STATEMENT.

The BUMP statement is used to increment the contents of a data-name by a value.

SYNTAX.

The syntactical structure of the BUMP statement is as follows:

```
BUMP  data-name-1  [ BY { data-name-2 } ] ;
                       { expression  }
```

DESCRIPTION.

Data-name 1 is incremented by the binary value of data-name 2 or the expression, and the sum is assigned to data-name 1.

NOTE

The sum is calculated on the low-order 24 bits of data-name 1, and any bits to the left are zero filled by the assignment operation.

If the BY option is omitted, a value of binary 1 is assumed.

The BUMP may also appear within an expression.

EXAMPLES.

Examples describing the use of the BUMP statement are as follows:

Examples	Comments
`BUMP X;`	Add 1 to X.
`BUMP X BY 4;`	Add 4 to X.
`BUMP X BY Z;`	Add the value of Z to X.
`A := BUMP X BY Z;`	Add the value of Z to X and assign the value to X. Then assign the value of X to A.

`IF (BUMP X BY Z) EQL ZERO`
`  THEN...;`
`  ELSE...;`
	Add the value of Z to X, assign the value to X, and then perform the comparison.

`BUMP A BY B := C;`	Assign the value of C to B and then add the value of C to A. Notice that C is added to A because of the replacement delete left part operator.
`X := BUMP A BY B := C;`	Replace B by the value of C, delete B, add C to A, and assign the value to A

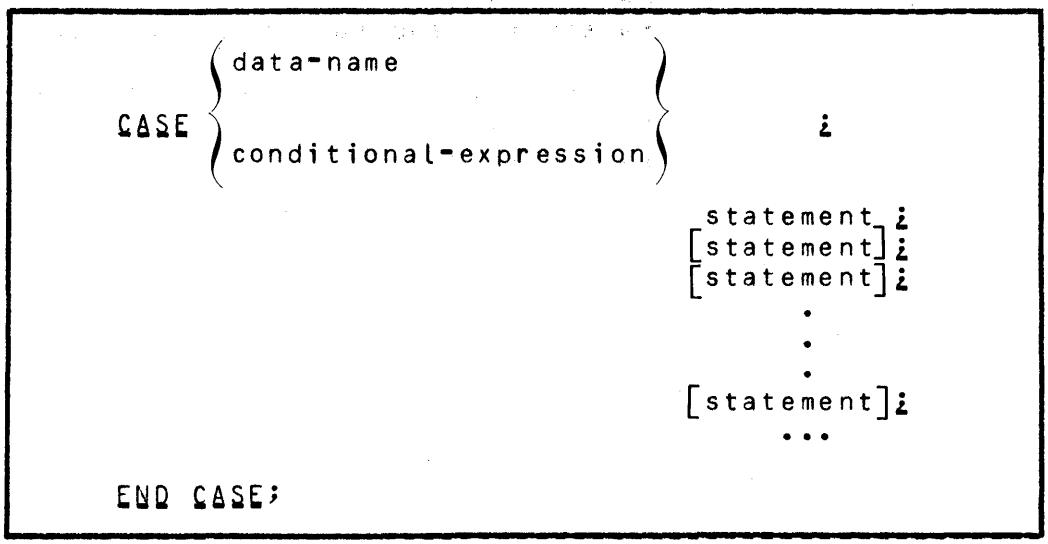
Examples	Comments
	and to X.
PROC.B (BUMP X);	Data-name X is bumped by 1 and then passed by name to procedure PROC.B.
PROC.B ((BUMP X));	The same as above except the pass is by value because of the extra set of parentheses.

CASE STATEMENT.

The CASE statement selectively executes only one statement within the CASE group of statements.

SYNTAX.

The syntactical structure of the CASE statement is as follows:



DESCRIPTION.

At execution time the data-name or conditional expression is valued as a binary. This value is used as a selector to choose from among the statements in the CASE group, such as a value of 2 selects the third statement. The statements in the group are numbered from 0 to N-1 for N statements. A negative value or a value greater than the number of statements in the CASE group causes an execution-time error.

All valid UPL statements, including nested CASE statements, DO-group statements, and IF...THEN...ELSE statements, are allowed and are counted as single statements.

After the selected statement has completed execution, program control passes to the statement immediately following the END CASE statement.

Null statements (refer to page 6-37) may be used to satisfy statement positions where no-operation condition is desired.

If a CASE statement is imbedded in a DO-group, then the execution of an UNDO statement terminates the DO-group and control passes to the end of the DO-group.

If a CASE statement is imbedded in a procedure, the execution of a RETURN statement (refer to page 6-40) passes control back to the statement that invoked the procedure.

NOTE

There exists an expression form of the CASE statement that has a different syntax and produces a different result (refer to page 8-4).

EXAMPLES.

Examples describing the use of the CASE statement are as follows:

Examples	Comments
CASE X; PROA; PROB; PROC; END CASE;	The value of X determines which procedure is called. X may vary in value from 0 through 2. If it is greater than the number of statements in the CASE statement, then a run-time interrupt occurs.
CASE (A * B) MOD 2; DO; IF X THEN UNDO X := X + 5; END; CASE X; PROK; PROL; . . . END CASE; END CASE;	The value of the expression is used to choose the statement to execute. A DO statement or a CASE statement is considered as one statement.

CHANGE STATEMENT.

The CHANGE statement is used to dynamically alter the attributes of a file during program execution.

SYNTAX.
The syntactical structure of the CHANGE statement is as follows:

```
CHANGE  internal-file-name IO (dynamic file attribute
        [[, dynamic file attribute ] ... ] ) ;
```

DESCRIPTION.
The CHANGE statement alters the attributes associated with a file. Any attributes declared in a FILE statement can be changed. The changes become effective when the file is OPENed. If the file is OPEN when the CHANGE statement is executed, it must be CLOSEd and re-OPENed to effect the changes.

Attributes not changed remain as originally declared.

NOTE
Refer to the FILE statement (page 7-7) for the default attributes associated with each device type.

DYNAMIC ATTRIBUTES.
The syntax for each of the dynamic attributes is as follows:

MULTI.FILE.ID

```
MULTI.FILE.ID := expression
```

The expression is concatenated with a 10-character string of EBCDIC blanks with the expression on the left. The left-most 80 bits (10 characters) then become the MULTI.FILE.ID.

If the expression is a bit-string, it will be left justified and right padded with spaces.

Examples.
Examples describing the use of the MULTI.FILE.ID attribute are as follows:

Examples	Comments
MULTI.FILE.ID := "MASTER"	The MULTI.FILE.ID is changed to "MASTER ".
MULTI.FILE.ID := @(4)FFF@	The MULTI.FILE.ID is changed to the hexadecimal string @(4)FFF4040404040404040@.

Examples	Comments
	Notice that this name could not be entered from the console printer (SPO); and this file cannot, therefore, be accessed by any SPO command.

FILE.ID

```
FILE.ID := expression
```

The expression is handled the same as the MULTI.FILE.ID expression.

Example.
An example describing the use of the FILE.ID attribute is as follows:

Example	Comment
FILE.ID := "PAYROLL"	The FILE.ID is now "PAYROLL ".

LABEL.TYPE

```
LABEL.TYPE := { BURROUGHS }
              { ANSI      }
              { UNLABELED }
```

DEVICE

```
DEVICE := expression
```

Where the expression must result in a bit-string of 10 bits, the hardware type (refer to the table below) is the low-order (right-most) six bits and the variant is the high-order (left-most) four bits.

The device, hardware type, and variants are as follows:

Device	Hardware Type	Variants
Invalid	0	
96-Column Punch	1	
80-Column Punch	2	Same as Printer
96-Column Reader/Punch	3	
96-Column MFCU	4	
96-Column Reader/Punch/Printer	5	
Paper Tape Punch	6	Same as Printer
Paper Tape Reader	7	
Printer	8	0 = Backup tape or disk 1 = Backup tape 2 = Backup disk 3 = Backup tape or disk 4 = Hardware only 5 = Backup tape only 6 = Backup disk only 7 = Backup tape or disk only 8 = Forms (Forms may be requested along with any of the above, 0-7.)
Invalid	9	
Read-Sorter	10	
Any Head per Track Disk (1A,1C,2B)	11	Same as any disk
Head per Track Disk (1A,1C)	12	Same as any disk
Head per Track Disk (2B)	13	Same as any disk
Only Disk Cartridge	14	Same as any disk

Only Disk Pack	15	Same as any disk
Disk Pack or Cartridge	16	Same as any disk
Any Disk	17	0 = Serial 1 = Random
96-Column Punch/Printer	18	
96-Column Reader	19	
Invalid	20	
80-Column Reader	21	
Console Printer (SPO)	22	
Invalid	23	
9-Track Tape (NRZ)	24	
7-Track Tape (NRZ)	25	
9-Track Tape (PE)	26	
7-Track Tape Cluster	27	
9-Track Tape Cluster	28	
9-Track Tape	29	
7-Track Tape	30	
Any Magnetic Tape	31	

PARITY.

```
PARITY := expression
```

The low-order bit of expression is interpreted as follows: 0 is ODD, 1 is EVEN.

TRANSLATION.

```
TRANSLATION := expression
```

The low-order three bits of expression are interpreted as follows: 000 is EBCDIC, 001 is ASCII, and 010 is BCL.

BUFFERS

```
BUFFERS := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of buffers.

LOCK

```
LOCK := expression
```

The low-order bit of expression is interpreted as follows: 0 is NOT LOCKed, and 1 is LOCKed.

OPTIONAL

```
OPTIONAL := expression
```

The low-order bit of expression is interpreted as follows: 0 indicates that a file must be present, and 1 indicates that a file is optional.

VARIABLE

```
VARIABLE := expression
```

The low-order bit of expression is interpreted as follows: 0 is fixed length, and 1 is variable length.

SAVE

```
SAVE := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of days this file is to be saved.

RECORD.SIZE

```
RECORD.SIZE := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of characters in a record.

RECORDS.PER.BLOCK

```
RECORDS.PER.BLOCK := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of records in a physical block.

REEL

```
REEL := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of reels of tape for this file.

NUMBER.OF.AREAS

```
NUMBER.OF.AREAS := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the maximum number of disk areas that can be opened for this file.

BLOCKS.PER.AREA

```
BLOCKS.PER.AREA := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the number of blocks of records in a disk area for this file.

PACK.ID

```
PACK.ID := expression
```

The expression is interpreted as the 10-character PACK.ID. It is handled the same as MULTI.FILE.ID.

SINGLE.PACK

```
SINGLE.PACK := expression
```

The low-order bit of expression is interpreted as follows: 0 is NO, and 1 is YES. The YES requires the file to be contained on one removable disk device.

ALL.AREAS.AT.OPEN

```
ALL.AREAS.AT.OPEN := expression
```

The low-order bit of expression is interpreted as follows: 0 is NO, and 1 is YES. The YES requires all areas of the file to be allocated at file-open time.

AREA.BY.CYLINDER

```
AREA.BY.CYLINDER := expression
```

The low-order bit of expression is interpreted as follows: 0 is NO, and 1 is YES. The YES requires each area of the file to begin on a cylinder boundary.

EU.SPECIAL

```
EU.SPECIAL := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the Electronic Unit (EU) of a head-per-track disk or the system drive number of a removable disk upon which the first disk area of this file is located.

EU.INCREMENTED

```
EU.INCREMENTED := expression
```

The low-order 24 bits of expression are interpreted as a positive number representing the increment to be added to the current EU or drive number for the location of the next disk area associated with this file. When the drive requested exceeds the number of reads-per-track EU's or the number of system drives, the next area is opened on the EU.SPECIAL drive.

USE.INPUT.BLOCKING.

```
USE.INPUT.BLOCKING
```

Specifies the record and block size are to be taken from the disk file header record.

SR.STATION.

```
SR.STATION = expression
```

Specifies which read station(s) is (are) to be used on a sorter reader file.

END.OF.PAGE ACTION.

```
END.OF.PAGE ACTION
```

Specifies the ON EOF statement is to be executed at the end of a page (channel 12) on the printer.

EXAMPLES.
Examples describing the use of the CHANGE statement with its various attributes are as follows:

	Examples	Comments
CHANGE	IN.FILE TO (DEVICE := @(2)00101@, MULTI.FILE.ID := "TEST", FILE.ID := "DISK", BUFFERS := 4, LOCK := @(1)@, RECORD.SIZE := 180, RECORDS.PER.BLOCK := 32, NUMBER.OF.AREAS := 8, BLOCKS.PER.AREA := 2, AREA.BY.CYLINDER := 1, SINGLE.PACK := 1, EU.SPECIAL := 0, EU.INCREMENTED := 1,);	IN.FILE is changed to DISK, SERIAL, test/DISK, with four BUFFERS, to be LOCKed, a record size of one disk segment, one full track per block with eight areas, two blocks per area, each new area at the beginning of a cylinder, starting on drive 0 for the first area with each additional area on a new drive.

CLEAR STATEMENT.

The CLEAR statement sets a data-name to a standard UPL-defined value.

SYNTAX.

The syntactical structure of the CLEAR statement is as follows:

```
CLEAR data-name  [ [, data-name-2] , ... ] ;
```

DESCRIPTION.

Data-names that are to be CLEARed must be arrays. The entire array is CLEARed. Paged arrays, however, cannot be CLEARed.

Multiple data-names can be specified and must be separated by commas.

A data-name DECLAREd as being type CHARACTER is CLEARed to spaces (hexadecimal 40). All other types are CLEARed to binary 0's. Paged arrays may not be CLEARed.

A semicolon must terminate the CLEAR statement.

CONDITIONAL INCLUSION STATEMENT.

The conditional inclusion statement conditionally includes UPL source code during compilation.

SYNTAX.
The syntactical structure of the conditional inclusion statement is as follows:

```
& IE [NOI] symbol-name [ [ { AND } [NOT] symbol-name-2 ] ... ]
                           { OR  }

   UPL-source statements

[& ELSE UPL-source statements]

 & END
```

DESCRIPTION.
The truth or falsity of the logical combination of the symbol-names determines the UPL-source statements that are included for compilation. If the result is true, the immediately following UPL statements are compiled. If the result is false and the & ELSE portion exists, the UPL statements following the & ELSE are compiled. The & END statement terminates the conditional inclusion statement.

The conditional inclusion statement is an UPL source language statement and can, therefore, be nested. Nested conditional inclusions cannot overlap; that is, each is matched with the most recent unmatched & IF statement.

EXAMPLES.
Examples describing the use of the conditional inclusion statement are as follows:

Examples	Comments
& SET SW1 SW2 SW3	The following statements are compiled into the program:
& RESET SW4 SW5	DECLARE (A,B,C,D,E,F,G,H) FIXED;
DECLARE (A,B,C,D,E,F,G,H) FIXED;	C := D;
& IF SW5	E := F;
A := B;	F := G;
& IF SW1	G := H;

Examples

Comments

```
B := C;
& END
& ELSE
C := D;
& IF SW4
D := E;
& ELSE
E := F;
& END
F := G;
& END
G := H;
```

CONDITIONAL PAGE STATEMENT.

The conditional page statement skips to the top of the next page on the compiler print-out.

SYNTAX.

The syntactical structure of the conditional page statement is as follows:

```
& PAGE
```

DESCRIPTION.

The conditional page statement is used with other conditional statements to control the compiler print-out.

The & (ampersand) must be in column 1. No semicolon is required at the end of the conditional page statement.

CONDITIONAL SYMBOL STATEMENT.

The conditional symbol statement defines and sets or resets symbols used in the conditional compiler statement.

SYNTAX.

The syntactical structure of the conditional symbol statement is as follows:

```
& { SET   } symbol-name-1 [ [symbol-name-2] ... ]
  { RESET }
```

DESCRIPTION.

The conditional symbol statement can appear anywhere in the UPL source language. The first occurrence of a symbol-name creates the symbol-name and allows any following conditional compiler statements to test its status (set or reset).

Symbol-names can be duplicates of data-names without causing syntax errors.

The scope of a symbol-name is from the first conditional symbol statement in which it occurs to the end of the input source code. That is, it can be referenced by any following statement without regard to lexic level boundaries.

The conditional symbol statement must contain an & (ampersand) in column 1 and be wholly contained in one card. Columns 72 through 80 of the card are for the sequence number.

NOTE

The semicolon is not part of the conditional symbol statement.

EXAMPLES.

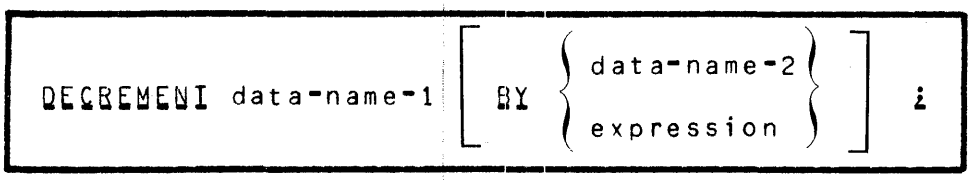
Examples describing the use of the conditional symbol statement are as follows:

Examples	Comments
& SET A	Set symbol-name A.
& SET A B C	Set symbol-names A, B, and C.
& RESET B	Reset symbol-name B.

DECREMENT STATEMENT

The DECREMENT statement is used to decrease the contents of a data-name by a value.

SYNTAX.
The syntactical structure of the DECREMENT statement is as follows:



DESCRIPTION.
Data-name 1 is DECREMENTed by the binary value of data-name 2 or the expression. If the BY option is omitted, a value of binary 1 is assumed. The contents of data-name 1 are permanently altered by the DECREMENT statement. If data-name 1 is larger than 24 bits, 0's are padded on the left.

The DECREMENT statement may also appear within an expression.

EXAMPLES.
Examples describing the use of the DECREMENT statement are as follows:

Examples	Comments
DECREMENT A;	Subtract 1 from A.
DECREMENT A BY 7;	Subtract 7 from A.
DECREMENT A BY B;	Subtract the value of B from A.
X := DECREMENT A BY B;	Subtract the value of B from A and assign the value of A to X.
IF (DECREMENT A BY B) EQL X THEN...; ELSE...;	Subtract the value of B from A and then compare A to X.
PROC.B (DECREMENT X BY A)	The data-name is DECREMENTed by the value of A and passed to the procedure PROC.B by name. An extra set of parentheses results in a pass by value.

DO STATEMENT.

The DO statement provides the capability to group a set of related statements together for programmatic control purposes.

SYNTAX.

The syntactical structure of the DO statement is as follows:

```
DO   [group-name]          [FOREVER] ;
                           statement;
                           statement;
                           statement ;
                                .
                                .
                                .
                           statement ;
END  [group-name] ;
```

DESCRIPTION.

The group-name option, if used, must be a unique name on its lexic level and must be the same in the DO statement and in its matched END statement.

A set of DO statement groups may be nested, but may not overlap. Every END statement is paired with the preceding unmatched DO statement, starting at the innermost set. An END statement is required for each DO statement grouping.

NOTE

For purposes of clarity, DO and END statements may be thought of as being a set of parentheses that surround a group of statements, thereby binding them as one statement for control purposes.

A DO statement through its END statement is considered as being a single statement. DO statement groups may be imbedded in CASE statement groups, IF statements, or another DO statement group. A maximum of 32 CASE, DO, or IF statements may be imbedded in any one nest. The UNDO statement, however, terminates up to a maximum of 16 nested DO statements.

DO, IF, and CASE statements define a code nesting level that is displaced under the column marked NL on the listing. Each nest must be wholly contained within its outer nest. That is, code nesting levels may not overlap.

The FOREVER clause implies that an unlimited iteration of the DO statement group occurs until an UNDO or a RETURN statement is executed. The execution of a RETURN statement causes control to be passed back from the PROCEDURE in which the DO statement is imbedded. The DO FOREVER option has a limit of 4096 bits of object code. If the FOREVER option is not used, no iteration of the DO statement group occurs.

EXAMPLES.
Examples describing the use of the DO statement are as follows:

Examples	Comments

`DO;`
`  BUMP SUM;`
`  DECREMENT DIFF;`
`  .`
`  .`
`  .`
`END;`
	The format of a DO statement requires the DO and a corresponding END.

`IF X EQL 0`
`  THEN DO;`
`  BUMP X;`
`  .`
`  .`
`  .`
`  BUMP SUM;`
`  END;`
`  ELSE DO OTHER;`
`  DECREMENT X;`
`  .`
`  .`
`  .`
`  BUMP SUM;`
`  END OTHER;`
	One of the two DO statements within the IF statement is executed, and then control is passed beyond the IF statement. The second DO statement is named OTHER, and its END statement must also contain the same name.

`DO THIS.ONE FOREVER;`
`  IF SUM LEQ ZERO`
`  THEN DO;`
`  SUM := SUM + 1;`
`  END;`
`  ELSE UNDO;`
`  END THIS.ONE;`
	The DO statement named THIS.ONE iterates until SUM is greater than 0. When that condition is reached, the UNDO statement following the ELSE terminates the DO statement.

`PROCEDURE ABC;`
`  DO ANY FOREVER;`
`  IF X GEQ 0`
`  THEN DO;`
`  DECREMENT X;`
`  BUMP SUM;`
`  END;`
`  IF SUM GEQ 0`
`  THEN UNDO;`
	This procedure contains several DO statements. The RETURN statement in the last IF statement also terminates the DO ANY statement by passing control out of PROCEDURE ABC.

Examples

```
            ELSE RETURN;
     END ANY;
         .
         .
         .
END ABC;
```

Comments

FINI STATEMENT.

The FINI statement signifies the end of source images to be compiled.

SYNTAX.

The syntactical structure of the FINI statement is as follows:

```
FINI;
```

DESCRIPTION.

The FINI statement is required and must be the last statement in the source program.

IF STATEMENT

The IF statement is used to conditionally execute one or two statements in a program.

SYNTAX.
The syntactical structure of the IF statement is as follows:

```
IF  conditional-expression  THEN statement-1 [ELSE statement-2]
```

DESCRIPTION.
The conditional-expression is evaluated, and the least-significant-bit of the result is interpreted as the controller bit.

If the ELSE clause is not specified, the controller bit is used to conditionally pass control to statement-1. If the controller bit contains the value of 1, then statement-1 is executed. If the controller bit is valued at 0, statement-1 is not executed and control passes to the statement immediately following the IF statement itself.

If the ELSE clause is specified, the controller bit is used to choose between statement-1 and statement-2. If the controller bit contains a value of 1, then statement-1 is executed and statement-2 is not executed. If it contains the value of 0 (zero), statement-1 is not executed and statement-2 is executed.

DO, IF, and CASE statements can be imbedded within an IF statement.

Each of these imbedded statements is called a nesting level, with a maximum of 32 levels allowed. The nesting-level number is given on the source code print-out under the NL column.

When using such nested IF statements, correspondence between the THEN and the ELSE statements must be maintained. That is, the innermost (highest NL number) ELSE is associated with the innermost THEN, and corresponding pairs continue outward (toward NL zero). The matching ELSE statement is required within nested IF statements. Null statements can, however, be used whenever a no-operation is desired.

Conditional-expression evaluation is performed from left-to-right in normal order unless parentheses are specified. The result of each operation is applied to the next operand until all operands are combined. The right-most bit of this result is the controller bit.

If all the operands are of type FIXED, all comparisons are signed FIXED. If any operand is not of type FIXED, then from that point on within the conditional-expression, comparisons are unsigned BIT.

If all the operands are of type CHARACTER, comparisons are from left-to-right. If any operand is non-CHARACTER then, from that point on within the conditional-expression, comparisons are type BIT.

The controller bit may itself be used as computational value (refer to the last example, page 6-35). A semicolon is not required with the IF statement because the subordinate statements end with them.

NOTE

There also exists an expression form of the IF...THEN...ELSE... statement that has a different syntax and produces a different result from the IF statement.

EXAMPLES.
Examples describing the use of the IF statement are as follows:

Examples	Comments
`IF X THEN A := B + C;`	If the data-name X contains a value whose least-significant (right-most) bit is a 1, the statement following the THEN is performed. Otherwise, control passes to the next sequential source statement.

NOTE

Execution time to choose the ELSE statement when coded is less than the time to choose the THEN statement.

`IF X EQL 0 THEN`
`  DO; A := B + C;`
`  BUMP X;`
`  END;`
	If X is equal to 0, the statement that follows the THEN is performed; otherwise, control passes to the next sequential statement. Notice that the statement that follows the THEN is a DO statement, which may itself contain several statements.

`IF X = 0 THEN A := B + C;`
`  ELSE A := X + Y`
`  XYZ := A;`
	If X equals 0, the statement that follows the THEN is performed. If X does not equal 0, the statement that follows the ELSE is performed. After one of the statements is executed, control is passed to the statement that follows the IF statement, that is, the XYZ := A; statement.

`IF SUBBIT (STRING, X)`
`  THEN CALL P1;`
`  ELSE CALL P2;`
	This is a conditional test of a bit string for the least significant bit in the string that is returned by the SUBBIT function. The rules explained in the preceding example apply, except that procedure P1 or P2 is then called.

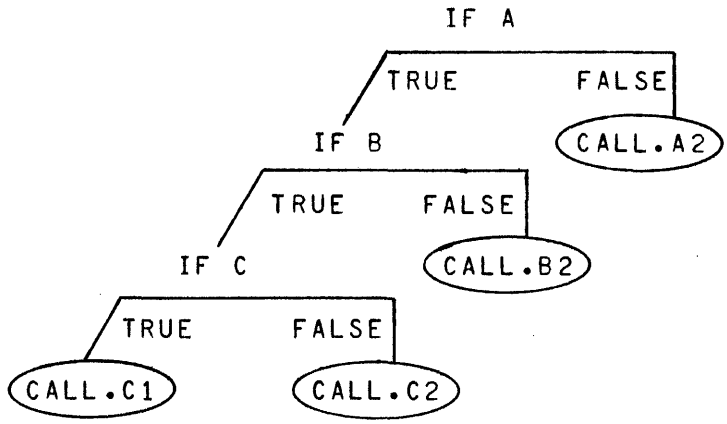
Examples

```
IF (A + B) = C THEN DO;
     SUM := SUM + 1;
     CALL P1;
     END;
  ELSE CASE C - A;
     SUM := SUM + 1;
     CALL P2;
     END CASE;
```

This is a conditional test of an expression. Subsequent processes depend on the outcome of the evaluation of the expression (A + B) = C. DO and CASE statement groups are allowed within IF statements.

```
IF A THEN
        IF B THEN
            IF C THEN CALL.C1;
                 ELSE CALL.C2;
          ELSE CALL.B2;
     ELSE CALL.A2;
```

Nested IF statements are allowed to any level. The associated THEN/ELSE pairs are defined as shown in the event tree.



NOTE
A better programming technique is to use DO and END statements around simple IF statements.

```
IF A OR B OR C THEN
        POSITIVE;
        ELSE NEGATIVE;
```

If A or B or C ends with the least-significant (right-most) bit a 1, the procedure POSITIVE is called. If A, B, and C all have a 0 (zero) in their least significant position, procedure NEGATIVE is performed.

```
IF (A GTR B) + (A LSS B)
     THEN X := 1;
     ELSE X := 0;
```

Each of the two conditional expressions is evaluated and returns a bit, 1 for true or 0 (zero) for false. The two bits are then added together, and the low-order bit of the result becomes the controller bit. In this example, if A equals B, X is set to 0 (zero); otherwise, X is set to 1.

LIBRARY STATEMENT.

The LIBRARY statement copies source language statements from the library into the program being compiled.

SYNTAX.

The syntactical structure of the LIBRARY statement is as follows:

```
& LIBRARY { file-name                          }
          { family-file-name/file-name         }
```

DESCRIPTION.

The source language images contained in the named file are copied into the program at the location of the LIBRARY statement.

The & (ampersand) must be in column 1.

No semicolon is required at the end of the LIBRARY statement.

The library file to be copied must be created in advance. The utility that creates the library files is invoked as follows:

```
a.  ? EX    UPL/LIBRARY
b.  ? FILE  DISK - family-file-name/file-name DISK SERIAL
c.  ? DATA
d.    Any UPL-source language statements
e.  ? END
```

NULL STATEMENT.

The null statement performs a no-operation function during object run-time.

SYNTAX.
The syntactical structure of the null statement is as follows:

DESCRIPTION.
Two adjacent semicolons are used to delimit a null statement.

The null statement is considered a complete statement, and it can be used whenever the syntax requires a complete statement. Its most common usage is in the CASE and IF statements to fulfill the syntax requirements, but not to perform operations. It also can be used in READ, WRITE, and SPACE statements.

The null statement can be used to control events within a compound IF statement; however, this control is more readily accomplished if DO/END statements are used within the compound IF statement sequence.

EXAMPLES.
Examples describing the use of the null statement are as follows:

Example	Comments
CASE decode; PRO.A; PRO.B; ; ; PRO.C; PRO.D; END CASE;	The data-name decode is used to select one of the six statements within the CASE statement body. If the value of decode is a 2 or a 3, no operation is performed.

PROCEDURE CALL STATEMENT.

The procedure call statement passes control to a regular (non-function) procedure. After the procedure has been completed, program control returns to the statement that follows the calling statement.

SYNTAX.
The syntactical structure of the procedure call statement is as follows:

```
procedure-name [ ( parameter [[,parameter]...] ) ];
```

DESCRIPTION.
A procedure call statement must always be a separate statement. That is, the procedure call statement **must never** appear adjacent to a replacement operator or within an expression.

The procedure being called must reside within range and cannot be a function procedure (refer to section 8).

Optional parameters must be separated by commas and can be comprised of data-names, literals, and function procedure designators, in any order. Evaluation of the parameter list is performed from left-to-right and from the innermost set of parentheses. Only a single name or value is passed for each parameter. The parameters passed at object run-time are matched from left-to-right with the parameter-names that are contained in the procedure head statement of the invoked procedure. The number of parameters that are passed **must** equal the number of names in the procedure head statement. The actual type (or length) passed and the corresponding FORMAL type (length) for each parameter **must** agree if the $FORMAL.CHECK compiler option has been used.

Parameters comprised of single data-names, array elements, or SUBSTRs that are not enclosed within an extra set of parentheses are passed-by-name. That is, the address, rather than a value, of the data-name is passed.

Passed-by-value are parameters comprised of literals, single data-names that are enclosed in an extra set of parentheses, the value returned from function procedures, or the result of any expression evaluation.

A value is not returned from a called procedure. If such a requirement exists, the result must be communicated through the use of global data-names or by passing the parameter by name and specifying the corresponding formal parameter in the procedure to the left of a replacement operator within an executable statement.

EXAMPLES.
Examples describing the use of the procedure call statement are as follows:

Examples	Comments
`PROX;`	PROCEDURE PROX is being invoked.
`IF X THEN PROX;` `  ELSE PROY;`	One of the two procedures is called depending on data-name X.
`IF ABC THEN AREA (L, W);` `  ELSE VOLM (L, W, H);`	One of two procedures is called. All of the parameters are being passed-by-name.
`AREA ((L), W * H);`	Both of the parameters are being passed-by-value.

RETURN STATEMENT

The RETURN statement transfers program control out of a procedure and back into, or immediately following, the invoking statement depending on the type of procedure being executed.

SYNTAX.
The syntactical structure of the RETURN statement is as follows:

Regular Format

```
RETURN;
```

Function Format

```
RETURN (expression) ;
```

DESCRIPTION.
The RETURN from a regular procedure passes control out of the procedure and back to the statement that follows the calling statement.

The RETURN from a function procedure also returns control and always returns a value to be used in place of the function designator within the invoking statement. The evaluation of the invoking statement then continues.

An expression must appear in the function procedure RETURN statement. The type (length) of the expression in the function procedure RETURN statement must agree with the type (length) option as contained in the procedure head statement if the $FORMAL.CHECK compiler option has been specified.

The function format used for a regular procedure results in a compiler error message.

The execution of a procedure END statement is the equivalent of a regular procedure RETURN statement or a function procedure RETURN statement containing a value of 0's.

EXAMPLES.
Examples describing the RETURN statement are as follows:

Examples	Comments
PROCEDURE ABC; . . . IF X THEN RETURN; .	The regular procedure ABC has several conditional RETURN statements. It also has an unconditional RETURN if none of the others is executed.

Examples

```
      .
      .
      IF Y THEN RETURN;
      .
      .
      .
      RETURN;
END ABC;
```

```
PROCEDURE XYZ FIXED;
      DECLARE (X, Y) BIT (16),
      Z FIXED;
      .
      .
      .
      IF X THEN RETURN (Z ::= X);
      .
      .
      .
      RETURN (14);
END XYZ;
```

The functional procedure has one conditional RETURN that calculates a value and passes it as a FIXED number. Notice that X and Y need not be of type FIXED. A second RETURN statement passes the value of 14.

```
PROCEDURE EXP (A, B) CHARACTER
VARYING;
      FORMAL (A, B) FIXED;
      DECLARE X FIXED;
            DO CAL FOREVER;
                  IF EQL 0 THEN
                  RETURN ("B IS" CAT
                  "ZERO");
            ELSE DO;
                  B := B * B;
                  DECREMENT A;
                  IF A EQL ZERO THEN
                  RETURN ("B IS" CAT
                  CONV (B, CHARACTER));
            END;
      END CAL;
END EXP;
```

The function procedure EXP calculates B to the A power. It RETURNs a VARYING length character string.

REVERSE.STORE STATEMENT.

The REVERSE.STORE statement is used to assign each data-name value in a list of data-names to the preceding data-name in the list. Also, it assigns an expression value to the last data-name.

SYNTAX.
The syntactical structure of the REVERSE.STORE statement is as follows:

```
REVERSE.STORE (data-name-1, data-name-2 [ [, data-name-3] ...],
               expression);
```

DESCRIPTION.
The value of data-name 2 is assigned to data-name 1, then the value of data-name 3 is assigned to data-name 2, and so on, until the value of the expression is assigned to data-name-n.

NOTE
Because each address in the data-name-list is calculated only once for the whole statement, no equivalent construct in UPL is as efficient.

EXAMPLES.
Examples describing the use of the REVERSE.STORE statement are as follows:

Examples	Comments
REVERSE.STORE (A,B,C,X+4);	The effect is the same as from the following statements: A := B; B := C; C := X+4; Notice that REVERSE.STORE (A,B,C,X+4); is not the same as A := B := C := X+4;
REVERSE.STORE (CASE N OF (A,B,C(I),D) EX+2);	This statement assigns the value EX+2 to one of the data-names. The one chosen depends on the value of n.

STOP STATEMENT.

The STOP statement terminates a program in an orderly or normal manner.

SYNTAX.
The syntactical structure of the STOP statement is as follows:

```
STOP;
```

DESCRIPTION.
A STOP statement can appear anywhere that an executable statement can appear.

Any number of STOP statements can be coded within the program, but only one is executed.

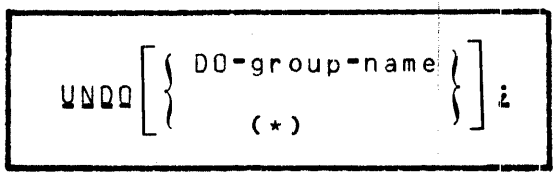
The UPL Compiler supplies a STOP statement as the last statement in the program if a STOP statement has not been specified.

When the STOP statement is executed all files are CLOSEd.

UNDO STATEMENT.

The UNDO statement provides the capability to transfer control out of DO statement groups.

SYNTAX.
The syntactical structure of the UNDO statement is as follows:



DESCRIPTION.
An UNDO statement is used only in conjunction with a DO statement. It passes program control to the statement that immediately follows the appropriate END statement for the specific DO statement group.

A simple UNDO statement (no options) passes control out of the current DO statement. The DO-group-name option passes control out of the named group. The asterisk passes control out of all nested DO statement groups within the procedure. A maximum of 16 DO-groups can be exited with one UNDO statement.

A DO statement group is subordinant to a procedure; therefore, control never passes out of a procedure with an UNDO statement. A RETURN statement, however, passes control from the procedure and terminates any DO-group in which it appears.

EXAMPLES.
Examples describing the use of the UNDO statement are as follows:

Examples

```
DO;
  .
  .
  .
IF A EQUAL B THEN UNDO;
  .
  .
  .
END;
```

Comments

A simple condition that ends the DO statement.

```
DO REPEAT FOREVER;
     IF SUM GEQ 0
          THEN DO;
               SUM := SUM -1;
               IF SUM GEQ 6
               THEN UNDO(*);
               END;
          ELSE IF SUM LSS 0;
               THEN SUM := SUM + 1;
```

Data-name SUM is tested for limits of + or - 5, then a message is printed. If the value is beyond the limits, all DO statements are ended.

Examples

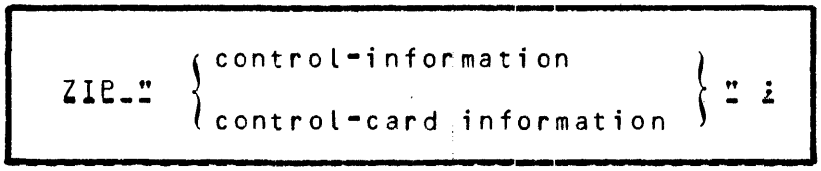
```
        ELSE DO;
             DECREMENT SUM;
             IF SUM LSS - 5
             THEN UNDO(*);
        END;
END REPEAT;
WRITE P.FILE ("SUM IS BETWEEN
              -5 AND +5");
```

Comments

ZIP STATEMENT.

The ZIP statement passes control information to the MCP as if it had been entered on the console printer (SPO).

SYNTAX.
The syntactical structure of the ZIP statement is as follows:



DESCRIPTION.
Any control information that can be entered via the console printer or the card reader can be used in a ZIP statement. The information passed must be surrounded by quotes.

EXAMPLES.
Examples describing the use of the ZIP statement are as follows:

Examples	Comments
ZIP "SO OPEN" ;	Sets the OPEN option in the MCP.
ZIP "EX CAL" ;	Begins the execution of program CAL.
ZIP "COMPILE PRINT UPL SYNTAX" ;	Program PRINT is to be compiled for SYNTAX only.
ZIP "SV LPA" ;	The MCP is requested to save the line printer.

SECTION 7

INPUT/OUTPUT STATEMENTS

GENERAL.
The input/output statements control the peripheral devices and READ or WRITE data external to the processor and its memory.

Input/output statements are listed alphabetically. The format of each statement is described in the following order:

a. Purpose.
b. Syntax.
c. Description.
d. Examples.

ACCEPT STATEMENT

The ACCEPT statement is used to input information from the console printer.

SYNTAX.
The syntactical structure of the ACCEPT statement is as follows:

```
ACCEPT  data-name [ , END.OF.TEXT ] ;
```

DESCRIPTION.
Data-name is the program area into which the information is moved. The information from the console printer is considered of type CHARACTER, and the assignment operator is used to move the data into the data-name. Therefore, truncation, padding, and left or right adjustment is in accordance with the rules of the assignment statement.

A maximum of 69 characters is allowed per ACCEPT statement.

The END.OF.TEXT option specifies that the END-OF-TEXT character (hexadecimal 03) is to be included with the message.

EXAMPLES.
Examples describing the use of the ACCEPT statement are as follows:

Examples	Comments

PROCEDURE A.MSG CHARACTER VARYING;
  DECLARE MSG CHARACTER (58);
RETURN (ACCEPT MSG);
  END A.MSG;
	Procedure A.MSG READs messages from the console printer and passes them back to their invoking statement. For example, MESSAGE := A.MSG; is a procedure call that can be used to input console printer message from anywhere in a program.

ACCESS.FILE.INFORMATION STATEMENT

The ACCESS.FILE.INFORMATION function returns two commonly required items from FILE PARAMETER BLOCK (FPB).

SYNTAX.

The ACCESS.FILE.INFORMATION syntax is as follows:

```
ACCESS.FILE.INFORMATION (internal-file-name { BIT       }
                                            { CHARACTER }
     , address-generating-expression) ;
```

DESCRIPTION.

The FPB is interrogated and the end-of-file (EOF) pointer and the device type are returned.

If the file is unopen and, therefore, the file information block (FIB) does not exist yet, the MCP ignores this communication.

The formating of returned data is as follows:

Item	Bit	Character
End-of-file pointer	24	8
Device type	6	2

The device types are described in the CHANGE statement. The data returned is assigned to the location as specified by the address-generating expression.

CLOSE STATEMENT.

The CLOSE statement releases control of a file from the program.

SYNTAX.
The syntactical structure of the CLOSE statement is as follows:

```
CLOSE  internal-file-name  [WITH  [CRUNCH  ,]  { REEL      }
                                               { RELEASE   }
                                               { PURGE     }
                                               { REMOVE    }
                                               { NO.REWIND }
                                               { LOCK      }

                           [IF.NOT.CLOSED]  ;
```

DESCRIPTION.
The internal-file-name must be the same as the file-name declared in the FILE statement.

The file must be in the OPEN state before it can be CLOSEd unless the IF.NOT.CLOSE option is used.

Files need not be explicitly CLOSEd. Memory space is immediately returned to the system whenever a file is CLOSEd, however, and the space can then be used for other purposes. Files that are closed at program termination are equivalent to CLOSE RELEASE.

The word WITH is optional, and its use has no effect.

CRUNCH applies to disk files that have only one disk area. The disk area allocated to the file is cut back to the actual size of the file. The word should be used only with files that can never be larger than when they are CLOSEd CRUNCH.

REEL specifies that the current reel of tape is to be CLOSEd, but the file is still open.

NO.REWIND inhibits the rewinding of a reel of tape.

RELEASE returns the memory file space to the system and does not enter file-names into the disk directory unless specified by the LOCK option in the FILE statement.

PURGE removes the file-name from the disk directory and returns the disk space to the disk-available table.

REMOVE removes a duplicate file-name from the disk directory if it is present and re-enters the name as referencing the new file.

LOCK enters the file-name into the disk directory.

IF.NOT.CLOSEd avoids an MCP termination of a program that attempts to CLOSE a file that is not OPEN.

The default is the same as RELEASE.

Files that are open at an abnormal program termination are CLOSEd with RELEASE.

If more than one option, excluding CRUNCH or IF.NOT.CLOSED, is requested, only the last is used.

EXAMPLES.
Examples describing the use of the CLOSE statement are as follows:

Examples	Comments
CLOSE HOLD WITH CRUNCH, RELEASE;	The file HOLD is to be cut back to its actual disk usage area, and its name is to be put into the disk directory.

NOTE
A file-name must be in the disk directory before another program can access it.

CLOSE MASTER;	The file MASTER is used as input, and its name is already in the disk directory. The file is no longer available to this program.
CLOSE OLD.MASTER WITH REMOVE;	The file OLD.MASTER is created by this program, and any other (duplicate) file by the same name is removed from the disk directory when this file-name is entered.

DISPLAY STATEMENT.

The DISPLAY statement causes a message to be printed on the console printer.

SYNTAX.
The syntactical structure of the DISPLAY statement is as follows:

```
DISPLAY   expression   [ , CRUNCHED ]   ;
```

DESCRIPTION.
The expression must be a data-name, literal, or character-string or must result in a printable message.

The CRUNCHED option removes all trailing blanks and substitutes one blank for each occurrence of multiple imbedded blanks.

EXAMPLES.
An example describing the use of the DISPLAY statement is as follows:

Example	Comments
PROCEDURE SEND.MSG (MSG); FORMAL MSG CHARACTER VARYING; DISPLAY "PLEASE" CAT MSG, CRUNCH; RETURN; END SEND.MSG;	Procedure SEND.MSG prints a message on the console printer and returns control to its calling procedure. For example, SEND.MSG ("LOAD FORMS") is a display message using the SEND.MSG procedure. The console printer outputs the message: PLEASE LOAD FORMS

FILE STATEMENT.

The FILE statement assigns an internal file-name to a physical input/output device and a list of attributes.

SYNTAX.

The syntactical structure of the FILE statement is as follows:

```
 FILE internal-file-name ( attribute-1  [, attribute-2] ... );

LABEL = { "family-file-name"
        { "family-file-name"/"file-name" }

LABEL-TYPE = { BURROUGHS }
             { ANSI      }
             { UNLABELED }

DEVICE = hardware-type [access-mode] [FORMS] [ [OR] BACKUP { TAPE } ]
                                                           { DISK }

MODE = [ { ODD  } ] [ { EBCDIC } ]
       [ { EVEN } ] [ { ASCII  } ]
                    { BCL    }

BUFFERS = integer

LOCK

OPTIONAL

VARIABLE

SAVE = integer

RECORDS = { characters-per-record                          }
          { characters-per-record/records-per-block        }

REEL = number-of-tape-reels

AREAS = number-of-disk-areas/number-of-records-per-area

PACK-ID = " literal "
```

```
OPEN  attribute-1   [ / attribute-2] ...

ALL.AREAS.AI.OPEN

AREA.BY.CYLINDER

SINGLE.PACK

EU.SPECIAL_= integer

EU.INCREMENTED_= integer

USE.INPUT.BLOCKING

SB.STATION_= integer

END.OF.PAGE.ACTION
```

DESCRIPTION.
The FILE statement options are described and illustrated in the paragraphs that follow.

FILE STATEMENT. The FILE statement is a declaration statement and must appear within the DECLARE portion of a program or a procedure. The file-name is the data-name by which the source program references the file. The option list must be surrounded by parentheses; all attributes are optional. A default status is set for omitted options, but varies by device.

LABEL OPTION. The LABEL option specifies the external file-name. It is the name the MCP uses to access information on an input/output device. The LABEL has two names, a family-file-name and a file-name, each enclosed in quotes and separated by a virgule. The family-file-name allows access to a multifile group, and the file-name allows access to a single file in the group ("PAYROLL"/"W2.SUMMARY"). Each name must be surrounded by quotation marks. The MCP uses only the first 10 characters of each name.

The file-name can be omitted. In such a case, the file is assumed to have no file-name. It is accessed by its family-file-name only.

The default for the LABEL option is setting a family-file-name the same as the internal-file-name and the file-name set equal to blanks.

Examples.
Examples describing the use of the LABEL option are as follows:

Examples	Comments
FILE TA.RECS (LABEL= "MASTER")	The internal data-name TA.RECS is declared, and the external file label MASTER is assigned.
FILE ERR (LABEL="MSG"/ "ERROR");	The data-name ERR is declared. An I/O file named ERROR of the multifile group MSG is being referenced.

LABEL.TYPE OPTION. The LABEL.TYPE option specifies the type of tape label. The default option is BURROUGHS.

DEVICE OPTION. The DEVICE option specifies the type of input/output peripheral on which the file resides. The input/output DEVICE types are as follows:

Device	Comment
PUNCH	80-column card
MULTI.FUNCTION.CARD	Any input/output functions on the 96-column card unit
PAPER.TAPE.PUNCH	
PAPER.TAPE.READER	
PRINTER	132-column
SORTER.READER	
DISK.FILE	Any head-per-track disk
DISK.FILE.1	1A and 1C head-per-track disk
DISK.FILE.2	2B head-per-track disk
DISK.PACK	Disk pack only
DISK.CARTRIDGE	Disk cartridge only
DISK.PACK.OR.CARTRIDGE	Any removable disk
DISK	Any disk
CARD	80-column card
SPO	Console printer

Device	Comment
TAPE.9.NRZ	
TAPE.7.UPRIGHT	
TAPE.9.PE	
TAPE.7.CLUSTER	
TAPE.9.CLUSTER	
TAPE	Any tape
TAPE.7	Any 7-track tape
TAPE.9	Any 9-track tape

The default option is TAPE.

ACCESS MODE OPTION. The ACCESS MODE specifies the ordering of disk record accesses. The options are SERIAL or RANDOM. The default is SERIAL.

FORMS OPTION. The FORMS option allows the operator to adjust the alignment of an output device. The FORMS option is applicable to PRINTER, PUNCH, or PAPER.TAPE.PUNCH only.

The default is FORMS omitted.

BACKUP OPTION. The BACKUP option specifies that an alternate output device can receive the information in the format of the primary device. The OR portion of the option specifies that the alternate device is to be used only if the primary device is unavailable.

The BACKUP option without the OR portion specifies the information must go to the alternate device.

The BACKUP option primary devices are PRINTER, PUNCH, PAPER.TAPE.PUNCH, and TAPE. The BACKUP alternate devices are TAPE and DISK.

The default is hardware only as specified in the DEVICE option.

Examples.
Examples describing the use of the DEVICE option and its parallel options are as follows:

Examples	Comments
FILE OUT.MASTER (DEVICE=PRINTER OR BACKUP DISK);	The FILE OUT.MASTER is printed if the printer is available. If the printer is unavailable, it goes to the DISK.
FILE W2.SUMMARY (LABEL= "PAYROLL"/"W2", DEVICE= DISK.PACK), FILE W2. REPORT (DEVICE=PRINTER FORMS OR BACKUP DISK);	Two files are declared, one with a label from the disk pack and a printer file with special forms. The printer file goes to disk if the printer is busy.

MODE OPTION. The MODE option specifies the parity and the code type of an input/output file. The options are ODD or EVEN parity and EBCDIC, ASCII, or BCL code. The default is ODD EBCDIC.

BUFFER OPTION. The BUFFER option specifies the number of input/output buffers to be allocated for the file. The default option is one buffer.

LOCK OPTION. The LOCK option requests the MCP to enter the file-name into the disk directory unless the file is closed with purge. Programs that are terminated abnormally by the MCP and have open disk files are entered into the directory. The default is NO LOCK.

OPTIONAL OPTION. The OPTIONAL option allows a program to execute without a file if the operator has responded to the NO FILE message with an OF console input message.

All reads or writes of an optional file that have an OF message execute the statement following the ON EOF (end of file).

VARIABLE OPTION. The VARIABLE option allows different length input/output records per READ/WRITE. The default option is fixed-size records.

SAVE OPTION. The SAVE option specifies the number of days a file is to be saved, that is, not to be destroyed. The default option is 30 days.

RECORDS OPTION. The RECORDS option specifies the number of characters in an unblocked record (physical size), or the number of characters per record (logical record size), and the number of records in the block (physical records per block).

Examples.
Examples describing the use of the RECORDS option are as follows:

Examples	Comments
FILE CARDSIN (RECORDS = 80);	The FILE CARDSIN contains 80-character unblocked records.
FILE TAPEOUT (RECORDS = 120/10);	The FILE TAPEOUT contains 10 records per block and 120 characters per record.

DEFAULT OPTIONS. The default options depend on the input/output DEVICE and have the following unblocked values:

Device	Unblocked_Value
CARD or PUNCH	80 characters
PRINTER	132 characters
DISK	180 characters
CONSOLE PRINTER (SPO)	72 characters
All others	80 characters

A character is the same as a byte.

REEL OPTION. The REEL number specifies the number of reels of tape on a file. The default option is 1.

AREAS OPTION. The AREAS option specifies the number of disk areas and the number of blocks (physical records) per area. The two values are separated by a virgule. Each physical record on the disk contains the number of characters as specified in the number of characters per block of the RECORDS option. The default option is 40 areas with 100 physical records per area.

PACK.ID OPTION. The PACK.ID specifies the name of the removable disk associated with this file. Only the first 10 left-most characters of the literal are used.

The default is 10 spaces, which implies the system disk.

OPEN OPTION. The OPEN option specifies the OPEN attributes to be associated with the file. As a result, an automatic OPEN is performed with the first input/output statement. If an OPEN statement is executed for the file, the attributes in the statement take precedence. The attributes are the same as in the OPEN statement, but here

must be separated by virgules.

Certain devices are defaulted OPEN as follows:

Device	Attribute
PRINTER	OUTPUT/NEW
CARD READER	INPUT
CARD PUNCH	OUTPUT/NEW
DISK	INPUT

ALL.AREAS.AT.OPEN OPTION. The ALL.AREAS.AT.OPEN option specifies that all requested disk space for a file opened NEW be allocated when the file is OPENed. The normal MCP procedure is to allocate each additional area as the file requires the space.

AREA.BY.CYLINDER OPTION. The AREA.BY.CYLINDER option specifies that each disk area begins on a cylinder boundary.

Default is disk space as required.

SINGLE.PACK OPTION. The SINGLE.PACK option specifies that the file resides completely on only one removable disk device.

Default is disk space as required.

EU.SPECIAL OPTION. The EU.SPECIAL option specifies which head-per-track electronic unit (EU) or systems DISK pack drive the file must be associated with. The possible range is 0 to 15.

Default is determined by the location of the first systems unit.

EU.INCREMENTED OPTION. The EU.INCREMENTED option is used with the EU.SPECIAL and specifies the EU or drive number that is incremented for each additional disk area allocated. The increment range is 0 to 15.

All areas of a file must be contained on systems disks. The increment number will wrap around when there are no more system units.

Default is 0 (zero).

USE.INPUT.BLOCKING OPTION. This option applies only to disk files. It specifies the record and block sizes are to be taken from the disk file header. That is, the record and block size attributes of the actual disk file are used.

The default is either the user specified attributes in the file statement or the default option of 180 character records unblocked.

SR.STATION OPTION. The number indicates which read station(s) is (are) to be used on a sorter-reader file. The possible stations are the magnetic ink character reader and the optical character reader. The read stations are interchangeable, and the systems documentation should be consulted for specific hardware configurations. Possible values are:

1 = first station
2 = second station
3 = both stations

The default is SR.STATION = 0.

END.OF.PAGE.ACTION OPTION. The END.OF.PAGE.ACTION causes the ON EOF statement to be executed at the end of a page on the printer.

The end of a page is detected as a channel 12 punch on the printer carriage control loop. The default option is the skip to channel 1 on the printer carriage control loop if channels 12 and 1 are punched; no action occurs if channel 12 is unpunched.

OPEN STATEMENT.

The OPEN statement establishes programmatic control of a data-file by requesting the MCP to make the data-file available to a program.

SYNTAX.
The syntactical structure of the OPEN statement is as follows:

```
OPEN  internal-file-name [WITH] attribute-1 [[, attribute-2]...];
```

The possible attributes are as follows:

a. INPUT
b. OUTPUT
c. NEW
d. LOCK
e. LOCK.OUT
f. NO.REWIND
g. REVERSE

DESCRIPTION.
The internal-file-name must be declared in a FILE statement. Use of the word WITH is for readability only. Multiple attributes must be separated by commas and have the following meanings or effects:

Attribute	Meaning/Effect
INPUT	A file exists and is to be read.
OUTPUT	A file exists and is to be written.
NEW	A file is to be created.
LOCK	This file cannot be written by any other program.
LOCK.OUT	This file cannot be accessed by any other program. NOTE LOCK and LOCK.OUT are true only for the duration of the locking program.
REVERSE	A tape is to be accessed in a backward direction.

Any files specified both INPUT and OUTPUT must be disk files.

No distinction is made between INPUT,OUTPUT or OUTPUT,INPUT declarations. Both imply a file exists and can be written to or read from.

OUTPUT, NEW creates a new file or a new version of an existing file. The CLOSE REMOVE option removes the old duplicate of an existing file.

If no attributes are specified, except for the card reader and the card punch that are defaulted OUTPUT, the default option is INPUT.

The FILE statement for a file being OPENed must be within scope.

Attributes that have been altered via the CHANGE statement are affected only during OPEN.

Attribute words, when used in an OPEN statement, cannot be DEFINEd.

The OPEN statement should be the first input/output statement executed for a file. That is, an OPEN statement should precede all READ, WRITE, and CLOSE statements that can be issued against a given file-name. The OPEN statement can be omitted only if the OPEN attributes are explicitly given in the FILE statement. A CLOSE statement for a given file must be executed before that file can be re-OPENed.

Because buffer-storage and file attributes are allocated when a file is OPENed, memory storage-area utilization can be significantly optimized by delaying the issuance of an OPEN statement until a file is actually needed. Also, when a file is no longer required by a program, the immediate execution of a file CLOSE statement optimizes memory storage-area utilization.

EXAMPLES.
Examples describing the use of the OPEN statement are as follows:

Examples	Comments
OPEN MASTER WITH INPUT;	The MASTER file is to be made available as input.
OPEN WORK WITH OUTPUT, INPUT, NEW;	The WORK file is a new file on disk to be used as output and as input.

READ STATEMENT.

The READ statement obtains an input record from a peripheral device as specified in the appropriate FILE statement.

SYNTAX.
The syntactical structure of the READ statement is as follows:

```
READ [LOCK]  internal-file-name

       [[record address expression]]  (data-name);

         [ON EOE executable statement;]

         [ON PARITY executable statement;]
```

DESCRIPTION.
The specified internal-file-name must be declared in a FILE statement and must be OPENed before a READ statement can be executed.

The LOCK option applies only to disk files and reserves a disk record for exclusive use by a program unit until a non-LOCK READ or WRITE statement is executed for the file.

The record-address-expression is applicable to random disk files only. Brackets ([]) are required. Random disk records are addressed by record-number displacement from the beginning of a file. The first record has a record-address of a 0 (zero), the second a 1 (one), etc., to n-1 for the nth record. The record-address-expression returns a binary value that is used to randomly access the record in the file.

The data-name is the receiving field for the information being READ. The internal-file-name is considered of type CHARACTER.

The replacement operator is used to move the information from the buffer to the data-name area. Truncation, padding, and left or right adjustment of the data is performed during the transfer.

The EOF and PARITY parts are considered subordinate to the READ statement and are, therefore, candidates for the special class of SEGMENT statements.

The reserved word ON is required before the EOF and PARITY options.

The EOF part specifies a single statement to be executed upon encountering the End-Of-File. If an EOF is detected and no EOF option is specified, the program is terminated.

The PARITY part specifies a single statement that is executed if a parity error occurs on the input/output device during the READ. If no PARITY part is specified, the normal MCP equivalent routines are exe-

cuted. The MCP can discontinue (DS) the program after trying n times to correct the situation.

The EOF and PARITY words are class III reserved words and can, therefore, be used as identifiers. If they are used as class III reserved words within the READ statement, null statements must be used to obtain the proper syntax. Reserved words, when used in the READ statement, may not be defined.

EXAMPLES.
Examples describing the use of the READ statement are as follows:

Examples	Comments
READ CARD.FILE(WORK); ON EOF STOP;	The CARD.FILE file is being READ into data-name WORK. When the End-Of-File is encountered the program terminates normally.
READ DISK.FILE [RANDOM.KEY] (RECORD); ON EOF PRO.END;	The DISK.FILE file is being accessed in a random manner under control of data-name RANDOM.KEY. The data READ is moved into data-name RECORD. At EOF or an invalid key, procedure PRO.END is invoked.

RECEIVE STATEMENT.

The RECEIVE statement is used to input a message to a data communications handler or another active program.

SYNTAX.
The RECEIVE statement syntax is as follows:

```
RECEIVE  address generates expression FROM program-name ;

         [ON Q.EMPTY executable statement;]

         [ON INVALID.REQUEST executable statement;]
```

DESCRIPTION.
The address-generating expression must contain the message.

The program name is the name of the SENDing program.

The ON Q.EMPTY statement is executed if there are no messages from the specified program.

The ON INVALID.REQUEST statement is executed if the MCP cannot recognize the request.

The MCP maintains queues in memory, if space exists, or on the disk.

SEARCH.DIRECTORY STATEMENT.

The SEARCH.DIRECTORY statement returns information, in the format specified, from the file header record on disk.

SYNTAX.
The SEARCH.DIRECTORY statement syntax is as follows:

```
SEARCH.DIRECTORY ( { PACK.ID CAI multifile-ID CAI file-ID }
                   { expression                           }

    , address generating expression , { BII       } ) ,
                                      { CHARACTER }

        [ON_FILE.MISSING executable statement;]

        [ON_FILE.LOCKED executable statement;]
```

DESCRIPTION.
The disk directory is searched for the named file and, if found, information is extracted from the file header record in the format specified and assigned to the address-generation expression location.

PACK.ID, multifile-ID, and file-ID must each be 10 characters long. If only multifile-ID is used, the PACK.ID and file-ID must be spaces.

Expression must result in a 30-character string, if used.

If the file is not present on disk, the statement following ON FILE. MISSING is executed.

If the file is open but with LOCK specified, the ON FILE.LOCKED executable statement is executed.

All values returned are in number of bits.

The format specifications of returned data are as follows:

Item-Name	Bit	Character
OPEN.TYPE	4	1
NO.USERS	8	2
RECORD.SIZE	24	4

Item-Name	Bit	Character
RECORDS.PER.BLOCK	24	4
EOF.POINTER	24	8
SEGMENTS.PER.AREA	24	8
ACCESS.DATE	16	6

The SEARCH.DIRECTORY statement is recommended in lieu of the ACCESS.FILE.HEADER statement because the data returned is not dependent on the MCP file header record layout.

Examples	Comments
SEARCH.DIRECTORY ("BBB TEST FILEX ", DATAXX, BIT);	A file named TEST/FILEX on a named user pack of BBB is being referenced.
SEARCH.DIRECTORY (" UPL " , SAVEINFO, CHARACTER);	The file UPL on the systems pack is being referenced.

SEEK STATEMENT.

The SEEK statement reads a random disk record into a buffer.

SYNTAX.
The syntactical structure of the SEEK statement is as follows:

```
SEEK [LOCK] internal-file-name

                [[record-address-expression]] ;
```

DESCRIPTION.
The SEEK statement replaces the automatic record-read of sequential files. It reads the record into the buffer from where the record is moved to the program space by the READ command.

The LOCK option reserves a record for exclusive use by a program until a non-LOCK READ or WRITE is executed for the file.

The internal-file-name must be declared in a FILE statement.

The record-address-expression returns a binary value that is used as the ordinal position of the record in the file. The first record is numbered 0 (zero), the second, 1 (one), etc., through the nth record, which is numbered n-1. Brackets ([]) around the record-address-expression are required.

The SEEK statement obtains the next record while the current one is being processed. Therefore, it often closely follows a READ statement.

EXAMPLES.
Examples describing the use of the SEEK statement are as follows:

Examples	Comments
SEEK D.FILE [NTH.REC];	The record specified by the NTH.REC key of the D.FILE file is found and loaded into a buffer.
SEEK D.FILE [5];	The sixth record of the D.FILE file is to be found and loaded into a main memory buffer.

SEND STATEMENT.

The SEND statement is used to output a message to a data communications handler or another active program.

SYNTAX.
The SEND statement syntax is as follows:

```
SEND  address generates expression IO program-name ;

      [ON Q.FULL executable statement;]

      [ON INVALID.REQUEST executable statement;]
```

DESCRIPTION.
The address-generating expression must contain the message.

The contents of the message are stored in a queue for the named program. Maximum message size is 65,535 bits, and maximum number of messages is 1023.

Control returns to the SENDing program, and the MCP will queue messages until the named program issues a receive.

The MCP maintains queues in memory, if space exists, or upon the disk.

The ON Q.FULL statement is executed if the queue has its maximum number of messages.

The ON INVALID.REQUEST statement is expected if the MCP for any reason cannot recognize the request.

SKIP STATEMENT.

The SKIP statement is used to control the carriage on the printer.

SYNTAX.

The syntactical structure of the SKIP statement is as follows:

```
SKIP  internal-file-name  TO  channel number ;
```

DESCRIPTION.

The SKIP statement causes the line printer to skip to the specified channel number on its carriage tape. The channel numbers are from 1 to 12.

EXAMPLES.

Examples describing the use of the SKIP statement are as follows:

Examples	Comments
SKIP P.FILE TO 1;	The P.FILE file must be an output file on the printer. The printer SKIPs to channel 1 (usually the top of a new page).
SKIP PRNT TO 12;	The printer SKIPs to channel 12 (usually at or near the end of a page).

SPACE STATEMENT.

The SPACE statement allows the user to skip over records in a sequential file.

SYNTAX.
The syntactical structure of the SPACE statement is as follows:

```
SPACE internal-file-name  [TO]  expression ;

  [ON EOF executable statement;]

  [ON PARITY executable statement;]
```

DESCRIPTION.
The internal-file-name must be declared in a FILE statement, and the file must be OPENed.

The expression returns a binary value that indicates the number of records to be skipped. If the value is negative, reverse or backward spacing is indicated.

The TO option specifies that spacing is in a forward or positive direction.

The ON EOF option specifies a statement that is to be executed if the EOF record is encountered while spacing.

If a parity error is detected, the ON PARITY option specifies a statement that is to be executed.

If the ON PARITY option is unspecified, the MCP enters its normal routines for parity errors. If the parity error is not corrected on successive retries, the program is discontinued (DS).

ON EOF and ON PARITY are class III reserved words that can be used as data-names. If they are used as data-names in the SPACE statement, null statements are required for proper syntax. When used as reserved words in the space statement they cannot be DEFINEd.

The ON EOF and the ON PARITY options are statements subordinate to the SPACE statement and can be segmented separately (refer to the SEGMENT statement, page 5-23).

EXAMPLES.
An example describing the use of the SPACE statement is as follows:

Example	Comments
SPACE TAPE.FILE TO X; ON EOF STOP;	The TAPE.FILE file is spaced the number of tape records specified by the binary value of data-name X. If the End-Of-File is encountered, the program is STOPped.

WRITE STATEMENT.

The WRITE statement transfers data from a specified output memory area to an assigned peripheral.

SYNTAX.

The syntactical structure of the WRITE statement is as follows:

```
WRITE [LOCK] internal-file-name [ NO       ]
                                [ SINGLE   ]
                                [ DOUBLE   ]
                                [ PAGE     ]
                                [ NEXT     ]
                                [ 1        ]
                                [ 2        ]
                                [ .        ]
                                [ .        ]
                                [ .        ]
                                [ 12       ]

      [[record-address-expression]] ( { character-string } );
                                    { data-name        }

       [ON EOF executable statement;]

       [ON EXCEPTION executable statement;]
```

DESCRIPTION.

A file must be OPENed before a WRITE can be executed. The LOCK option reserves a disk record for exclusive use of a program until a non-LOCK READ or WRITE is executed for the file.

The internal-file-name must be declared in a FILE statement.

The first option is used for printer control. The NO, SINGLE, and DOUBLE options specify lines of paper movement. The PAGE option spaces paper to the top (channel 1) of the next page. The NEXT option spaces paper to the next channel punch. The numbers 1, 2, ... 12 space paper to the specified channel punch.

The record-address-expression returns a binary value that is used as the number of the record being written to on a random disk file. The records are numbered in sequence from 0 to n-1 for an n record file on disk. Brackets ([]) around the record-address-part are required.

The data-name or character-string option is the program area from which the record is written.

The ON EOF executable statement is executed at the end of available space on disk.

Execution of the ON EXCEPTION executable statement is dependent upon the peripheral device (refer to the MCP manual).

The EOF and the EXCEPTION options are subordinate to the WRITE statement and can be segmented (refer to the SEGMENT statement).

EOF and EXCEPTION are class III reserved words that can be used as data-names. If they are used as data-names within a WRITE statement, null statements may be required for proper syntax. When they are used as reserved words in the WRITE statement, they may not be DEFINEd.

If the END.OF.PAGE.ACTION file attribute has been specified in the FILE statement and an end of page (channel 12 punch on the printer carriage control tape) is detected, then the ON EOF statement is executed. This facilitates, for example, printing totals or headings without the necessity of a line counter.

EXAMPLES.
Examples describing the use of the WRITE statement are as follows:

Examples	Comments
WRITE P.FILE (REC);	The REC record is written to the P.FILE file. The length of the output is coded in the FILE statement.
WRITE D.FILE [NTH.REC] (WORK); ON EOF PRO.END;	A disk file named D.FILE is written in a random mode with NTH.REC as the KEY. The data-name WORK is outputted. At the End-Of-File procedure PRO.END is called.

SECTION 8

FUNCTIONS

GENERAL.

UPL-supplied functions are a set of procedures that are incorporated directly into the language to facilitate ease of use and speed of execution.

The usage of supplied functions is similar to invocation of a function procedure written by a programmer. Such functions always return or reference a value; therefore, functions supplied by UPL are expressions.

UPL functions can be divided into three groups.

The first group is exactly like a user function procedure. The appearance of its name in an expression is replaced, at run-time, by a value. The following are examples of the first group:

a. CONVERT.
b. BINARY.
c. LENGTH.
d. CAT.

The second group is more akin to an input/output statement because it requests information from the operating system (MCP). The following are examples of the second group:

a. TIME.
b. DATE.
c. NAME.OF.DAY.

The third group is similar to a function procedure except that it returns an address rather than a value. It may, therefore, appear to the left of an assignment or replacement operation. The third group is in actuality an address-generating expression, but it is included here for convenience. The following are examples of the third group:

a. SUBBIT.
b. SUBSTR.

BASE.REGISTER FUNCTION.

The BASE.REGISTER function returns the absolute main memory address of the beginning of the data space of the program.

SYNTAX.

The BASE.REGISTER function syntax is as follows:

```
BASE.REGISTER
```

DESCRIPTION.

The BASE.REGISTER function returns a 24-bit value that is the current absolute main memory address of the beginning of the data space of the program.

In a multiprogramming environment two separate executions of BASE.-REGISTER may not yield the same results because the MCP may have moved the data of the program to a new location in memory.

BINARY FUNCTION.

The BINARY function converts a character-string to a sign and a 23-bit value of type FIXED.

SYNTAX.
The syntactical structure of the BINARY function is as follows:

```
BINARY (expression)
```

DESCRIPTION.
The expression must result in a character-string of eight or fewer decimal digits. Truncation occurs on the left for any strings greater than eight characters.

The character-string is assumed to contain decimal characters. Only the low-order (right-most) four bits of each character are used during the BINARY function; that is, the zone bits are ignored.

The character-string is converted to a FIXED number. Decimal values in excess of 8,388,608, but less than 16,777,215 (or any multiple of this range), cause the FIXED value to appear negative. Binary lengths in excess of 24 bits are truncated on the left. The DECIMAL function is the opposite of the BINARY function.

EXAMPLES.
Examples describing the use of the BINARY function are as follows:

Examples	Comments
A := BINARY (XYZ);	Data-name XYZ is assumed to contain a character string of eight or fewer numeric characters. The character-string is converted to a binary value and is assigned to A.
B := BINARY ("255");	B now contains 11111111.

CASE FUNCTION.

The CASE function is used to conditionally evaluate one expression from among a list of expressions.

SYNTAX.
The syntactical structure of the CASE function is as follows:

```
CASE expression-1 OF (expression-2 [ [, expression-3] ... ] )
```

DESCRIPTION.
The value of expression-1 is used as the ordinal position of the expression in the list to be executed. The first expression in the list is 0 (zero). The value of the executed expression is the value of the CASE function. A range check is performed, and an out-of-bounds value for expression-1 causes termination of the program. Notice that the CASE expression differs from a CASE statement because the CASE expression returns a value.

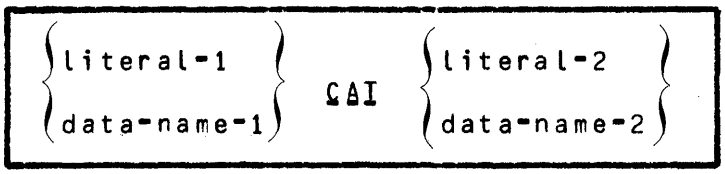
EXAMPLES.
Examples describing the use of the CASE function are as follows:

Examples	Comments
`X :=CASE A OF (B, C, D, E);`	The value assigned to X is dependent upon the value of A. For example, if A = 0 then X := B, or if A = 1 then X := C, or if A = 2 then X := D, or if A = 3 then X := E.

`Z := A + CASE N OF (X, Y, X + Y, X - Y) *2;`
`FOR A = 5, N = 2, X = 3,`
`and Y = -4`
Z is replaced by the value 3.
	Evaluation is Z := 5 + (3 + (-4) * 2).

CAT FUNCTION.
The CAT function programmatically concatenates two strings of data and forms a new string.

SYNTAX.
The syntactical structure of the CAT function is as follows:



DESCRIPTION.
Data items can be linked together (concatenated) by using the CAT function. Although this function is intended to concatenate bit-strings or character-strings, it can be used with any combination of data-types. The limit of data-items that can be concatenated is 8000 characters or 8000 bits.

The CAT function can specify, within the above limit, several operands connected by the required number of CAT functions.

If the operands are defined as being character, the result of a CAT function operation is a string of characters. For any other combination of operand data-types, the result is a string of bits.

EXAMPLES.
Assume the following declaration and initializations:

```
Declare A character (1), B bit (3), C fixed, X bit (6),
Y character (2), Z bit (11), XX bit (27).
```

Examples	Comments
A = "B"	Data-name A comprises a character-string containing the letter B.
B = @(1)101@	Data-name B comprises a bit-string that contains the binary value of 5. The length of the data-string is three bits.
C = +10	Data-name C comprises a FIXED-string that contains the positive (+) decimal value of 10.

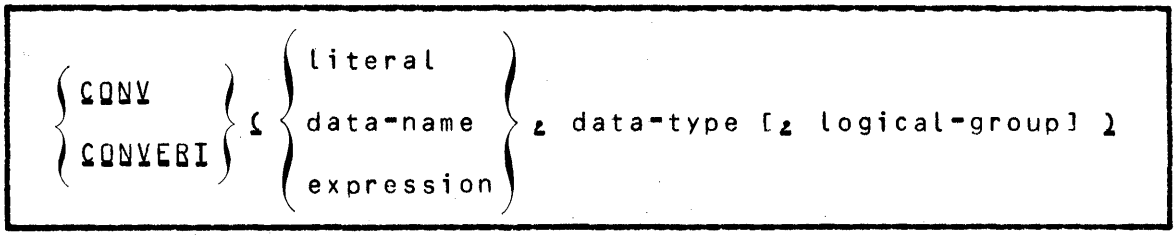
The contents of data-names A, B, and C are known; therefore,

X := B CAT B;	A binary value of 45; that is, @(1)101101@ is created. The length of the data-string is six bits. The result of the concatenation is assigned to data-name X.
Y := A CAT A;	A character-string, comprised of two bytes, that

Examples	Comments
	has the value of "BB" is created. The value is assigned to data-name Y.
Z := A CAT B;	A binary value of 1557; that is, @(1)11000010101@ is created. The length of the data-string is 11 bits. The result of the concatenation is assigned to data-name Z.
XX := B CAT C;	A binary string equivalent to the UPL octal notation @(3)500000012@ is created. The result of the concatenation is assigned to data-name XX.
X := A CAT B := 4;	The CAT function is lower in precedence than the := function. Data-name B is therefore set to a value of 4 before B is concatanated with data-name A. The result of the CAT is then assigned to data-name X. The second example describing the usage of the CONVERT function is also an example of the CAT function (refer to page 8-10).

CONVERT FUNCTION.
The CONVERT function facilitates the conversion of one data type to another.

SYNTAX.
The syntactical structure of the CONVERT function is as follows:



DESCRIPTION.
The entry of a data-name, literal, or expression denotes the data-item that is converted to the specified data type or logical group.

The data-type clause is required. It defines the output conversion data-type. The data-type is defined as follows:

a. FIXED.
b. CHARACTER.
c. BIT.

The logical-group clause is required only when converting from type BIT to type CHARACTER or from type CHARACTER to type BIT. It specifies the number of bits (of the bit string) that correspond to a character in the character string. The bit-groups specified are as follows:

Logical-Group	Comments
1	Bit-grouping is in binary representation.
2	Bit-grouping is in quartal representation.
3	Bit-grouping is in octal representation.
4	Bit-grouping is in hexadecimal representation.

If no bit-grouping is indicated, 4 (hexadecimal) is assumed. All truncation or padding between strings of unequal lengths is performed according to the rules as outlined in the assignment statement.

The conversion of data from type BIT to type CHARACTER expands the specified logical bit-grouping into a character (byte) format by prefixing 0's (zeros) to the most significant positions, and it should not be construed as being an EBCDIC conversion of the data. Therefore, the

conversion does not return printable decimal numbers. The result merely represents eight bits of data for further manipulation as may be programmatically desired.

To convert from type BIT to printable characters, first convert type BIT to type FIXED and then type FIXED to type CHARACTER.

The conversion of data from type FIXED to type CHARACTER results in a sign and seven printable (EBCDIC) decimal numbers; leading printable 0's (zeros) are not suppressed.

The conversion from type CHARACTER to type FIXED is performed in the following manner:

a. The type CHARACTER data-name is scanned from left to right until a sign or non-space character is encountered.

b. If a sign is encountered, it is noted and removed.

c. After encountering a sign or non-space character, only the right-most seven characters of the data-name are converted.

d. The low-order four bits of each character are considered a binary value times a power of 10 for each position from the right. The high-order four bits are ignored. Decimal values in excess of 8,388,607 positive or 8,388,608 negative have the 2 raised to the 24th power bit ignored.

If the sign as previously noted is negative, the FIXED number is expressed in the complement form of 2.

The various forms of data conversion are briefly described in the following chart.

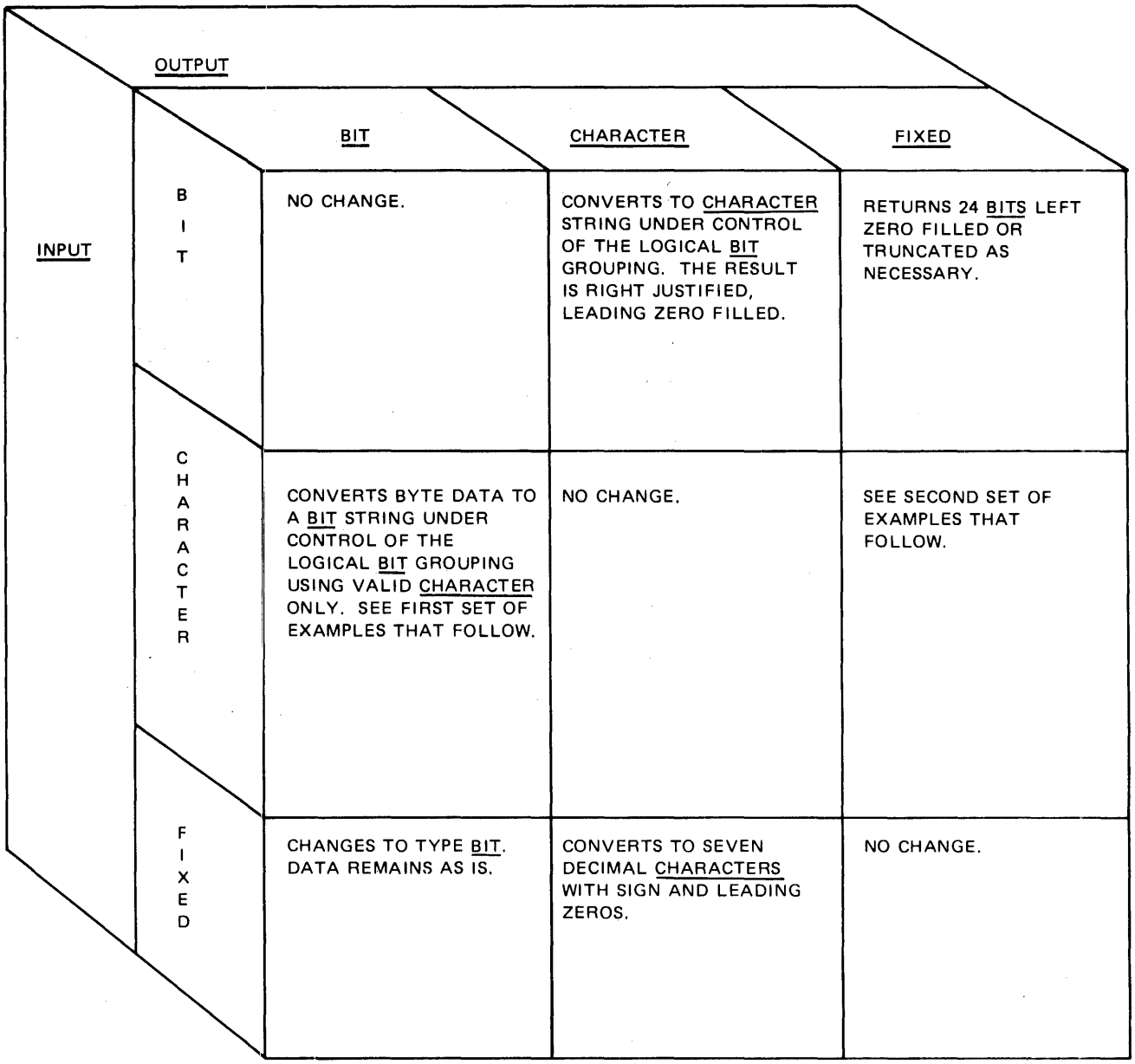


Figure 8-1. Data Type Conversion Chart

EXAMPLES.
Assume that data-name CX contains a character whose binary value is 00000111, and data-name B is declared type BIT (4).

Examples	Comments
B := CONV (CX,CHARACTER,4);	The contents of data-name B contain the hexadecimal value 7 (0111).
B := CONV (CX,CHARACTER,3);	The contents of data-name B contain the octal value of 6 (0110). Only the right-most three bits of data-name CX are assigned to B.

Assume data-name CARD contains the characters + 4095, and FX is of type FIXED.

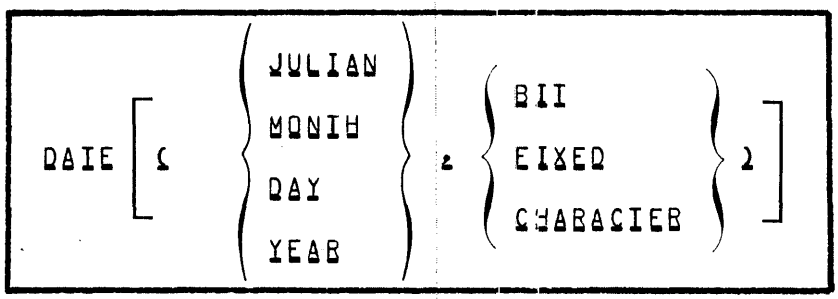
Examples	Comments
FX := CONV (CARD, FIXED);	The contents of FX contain hexadecimal 0007FF.
DECLARE N FIXED, B BIT (8);	
N := +5;	Data-name N contains the value +00...101 at object run-time.
B := @BC@;	Data-name B contains the hexadecimal value BC (binary value 1011 1100) at object run-time.
OUTPUT := "ENTRY NO." CAT CONV (N,CHARACTER) "IS" CAT CONV (B,CHARACTER, 2);	This statement produces a data-string object run-time in the form of: "ENTRY NO. + 0000005 IS 2330."

In the preceding example, the literal value "ENTRY NO.", the result of converting data-name N, the literal value "IS", and the result of converting data-name B are made into a continuous string of data by the insertion of concatenation (CAT) function designators. The result of converting the FIXED value contained in data-name N to a printable character is +0000005 with no suppression of 0's (zeros) or arithmetic sign. The result of converting the bit value contained in data-name B, when using the character-to-quartal syntax as specified, is as follows:

a.	10	11	11	00	(binary).
b.	2	3	3	0	(quartal).
c.	F2	F3	F3	F0	(hexadecimal character).

DATE FUNCTION.
The DATE function returns a string that contains the current month, day, year, or Julian date.

SYNTAX.
The syntactical structure of the DATE function is as follows:



DESCRIPTION.
DATE without the option is the same as DATE (MONTH, CHARACTER).

The format of each option and the lengths of the strings are as follows:

Option	Format	Bit	Digit Length	Characters
JULIAN	YY/DDD	7/9	2/3	2/3
MONTH	MM/DD/YY	4/5/7	2/2/2	2/2/2
DAY	DD/MM/YY	5/4/7	2/2/2	2/2/2
YEAR	YY/MM/DD	7/4/5	2/2/2	2/2/2

Notation used in the preceding table is as follows:

a. YY equals the year, DD or DDD equals the day, MM equals the month.

b. Digits are equal to four bits, that is, two decimal digits per byte.

c. Characters are equal to eight bits.

DECIMAL FUNCTION.

The DECIMAL function converts the right-most 24 bits of an expression from a binary value to a character-string.

SYNTAX.
The syntactical structure of the DECIMAL function is as follows:

```
DECIMAL (expression-1 , expression-2)
```

DESCRIPTION.
The right-most 24 bits or less of expression-1 are converted from a binary value to a decimal character-string equal to the number of characters in length requested in expression-2. No more than eight characters are produced. If the decimal character-string is greater than the length requested in an expression-2, truncation occurs on the left. If the character-string is less than expression-2, hexadecimal (00) zeros are padded on the left.

EXAMPLES.
Examples describing the use of the DECIMAL function are as follows:

Examples	Comments
X := DECIMAL (A, 4);	Data-name A is converted from a 24-bit binary value to a 4-character numeric string.
z := DECIMAL (@FF@, 3);	Z now contains the character-string 255.

HEX.SEQUENCE.NUMBER FUNCTION.

The HEX.SEQUENCE.NUMBER function allows the hexadecimal equivalent of the sequence number of the source language statement to be referenced at run-time.

SYNTAX.

The syntactical structure of the HEX.SEQUENCE.NUMBER function is as follows:

```
HEX.SEQUENCE.NUMBER
```

DESCRIPTION.

The HEX.SEQUENCE.NUMBER results in a bit-string of eight hexadecimal digits that represents the source-language line number being compiled.

EXAMPLE.

An example of the use of the HEX.SEQUENCE.NUMBER function is as follows:

Example	Comments
X := HEX.SEQUENCE.NUMBER 12753000	The value assigned to the data-name X at run-time is @(4)12753000@.

IF FUNCTION.
The IF function is used to conditionally evaluate one expression from a set of two.

SYNTAX.
The syntactical structure of the IF function is as follows:

```
IF  expression-1  THEN  expression-2  ELSE  expression-3
```

DESCRIPTION.
If the value of expression-1 is TRUE, that is, the least-significant-bit is a 1, the value of the expression that follows the THEN becomes the value of the IF function. If the value of expression-1 is FALSE, the value of the IF function is the value of expression-3. Notice that the IF function differs from an IF statement because it is an expression rather than a statement. It results in a value that must be used in a larger expression.

EXAMPLES.
An example describing the use of the IF function is as follows:

Example	Comments
X := IF B MOD 2 THEN "ODD" ELSE "EVEN";	Data-name X is assigned the word ODD if B is an odd number. If B is an even number, data-name X is assigned the word EVEN.

LENGTH FUNCTION.

The LENGTH function returns the expression length in a 24-bit type BIT value format.

SYNTAX.
The syntactical structure of the LENGTH function is as follows:

```
LENGTH  (expression)
```

DESCRIPTION.
If the expression returns a character-string, the LENGTH is the number of characters; otherwise, the LENGTH is the number of bits.

EXAMPLES.
Examples describing the use of the LENGTH function are as follows:

Examples	Comments
X := LENGTH (ABC);	The length of the data named by ABC is assigned to X.
X := LENGTH ("WARM");	X contains a binary value of 4.

LIMIT.REGISTER FUNCTION.

The LIMIT.REGISTER function returns the main memory limit address of the data space of a program.

SYNTAX.

The LIMIT.REGISTER function syntax is as follows:

```
LIMIT.REGISTER
```

DESCRIPTION.

The LIMIT.REGISTER function returns a 24-bit value that is the absolute main memory address of the data space of a program.

In a multiprogramming environment two successive executions of the LIMIT.REGISTER function may not yield the same results because the MCP may have moved the data of the program to a new location in memory.

MEMORY SIZE FUNCTION.
The MEMORY SIZE function returns the size of the requested available memory.

SYNTAX.
The MEMORY SIZE function syntax format is:

```
{ M.MEM.SIZE }
{ S.MEM.SIZE }
```

DESCRIPTION.
The requested memory size is returned as a 24-bit binary value indicating the number of bits in the memory.

S.MEM.SIZE is the size of all installed main memory including any being utilized as control memory.

M.MEM.SIZE is the size of high-speed control memory installed in the processing unit.

MOD_FUNCTION.
The MOD function results in the remainder of a divide.

SYNTAX.
The syntactical format of the MOD function is as follows:

DESCRIPTION.
The MOD function returns a value that is the remainder of a divide. If both the divisor and the dividend are type FIXED, the MOD returns a FIXED value.

If either or both are not type FIXED, then the MOD returns a positive 24-bit value.

EXAMPLE.
An example describing the MOD function is as follows:

Example	Comments
X Y MOD 52	The value assigned to X is in the range 0 to 51.

NAME.OF.DAY FUNCTION.

The NAME.OF.DAY function is a class II reserved word that returns a character-string for the name of the day.

SYNTAX.

The syntactical structure of the NAME.OF.DAY function is as follows:

DESCRIPTION.

The returned character-string is nine characters long, left-justified with trailing blanks.

SEARCH.LINKED.LIST FUNCTION.

The SEARCH.LINKED.LIST searches a predefined structure for a true condition.

SYNTAX.

The syntactical structure of the SEARCH.LINK.LIST function is as follows:

```
SEARCH.LINKED.LIST ( expression-1 , expression-2 , expression-3 ,
                     relational , expression-4 )
```

DESCRIPTION.

The predefined structure is searched for a true condition or until the end of the structure. If a true condition is found, the base relative address of the substructure is returned. If the search fails, @FFFFFF@ is returned.

Expression-1 is the base relative address of the first substructure to be examined.

Expression-2 is the relative offset (in bits) in the substructure of the 24-bit field being compared with expression-3.

Expression-3 is the 24-bit data being compared with a field in the substructure.

The relational is one of the relational operators, that is,

EQL	=	GTR	>
NEQ	≠	LEQ	≤
LSS	<	GEQ	≥

Expression-4 is the relative offset in the substructure of the 24-bit field containing the base relative address of the next substructure to be examined if the comparison fails.

NOTE

The SEARCH.LINKED.LIST function is used by the MCP to allocate memory space.

SEQUENCE.NUMBER FUNCTION.

The SEQUENCE.NUMBER function allows the character equivalent of the source language line number being compiled to be used in the program.

SYNTAX.

The syntactical structure of the SEQUENCE.NUMBER function is as follows:

```
SEQUENCE.NUMBER
```

DESCRIPTION.

The SEQUENCE.NUMBER function results in a character-string of eight EBCDIC characters that represent the sequence number of the current source language statement being compiled.

EXAMPLE.

An example describing the use of the SEQUENCE.NUMBER function is as follows:

Example	Comment
Z := SEQUENCE.NUMBER 01234500	The character string "01234500" is assigned to data-name Z.

SUBBIT FUNCTION.
The SUBBIT function provides the capability to address one or more data-bits within a data-name.

SYNTAX.
The syntactical structure of the SUBBIT function is as follows:

```
SUBBIT (data-name-1, { literal-1    } [, { literal-2    } ] )
                      { data-name-2 }    { data-name-3 }
                      { expression-1}    { expression-2}
```

DESCRIPTION.
Data-name 1 is considered a data-name of type BIT regardless of its previous declaration.

Data-name 2, literal-1, or expression-1 at object run-time is evaluated as a positive number that is used as the ordinal position of the first bit to be accessed within the specified bit-string. The most significant bit (left-most) within a bit-string is bit 0.

Data-name 3, literal-2, or expression-2 is evaluated as a positive number that is used as the number of bits to be accessed within the bit-string.

The omitting of data-name 3, literal-2, or expression-2 results in the accessing of a string from the bit specified by the value of data-name 2 through the last bit in the string.

A range-check is performed on data-name 2 and data-name 3, and an out-of-bounds value causes an interrupt of the program. That is, data-name 2 must point into the string, and data-name 3 must not specify more bits than exist between the first bit being accessed and the end of the string.

A resultant value of 0 (zero) for data-name 3 is valid and results in no accessing of the data.

If a SUBBIT function appears to the left of a replacement operator, it is treated as a data-name. Truncation, fill, and data alignment are performed by the operator with type BIT being the destination field-type. If data-name 2 or data-name 3 is declared as being of type CHARACTER, it is evaluated as being a binary number. That is, if a value of 1 is given, it is equal to the internal EBCDIC value of 11110001, which converts to a decimal representation of 241, which results in the accessing first of the 241st bit within the string.

The SUBBIT function may be passed to a procedure, by name or by value, according to the following conventions:

a. Statement SUBBIT (data-name 1, data-name 2, data-name 3) is defined as being a pass-by-name.

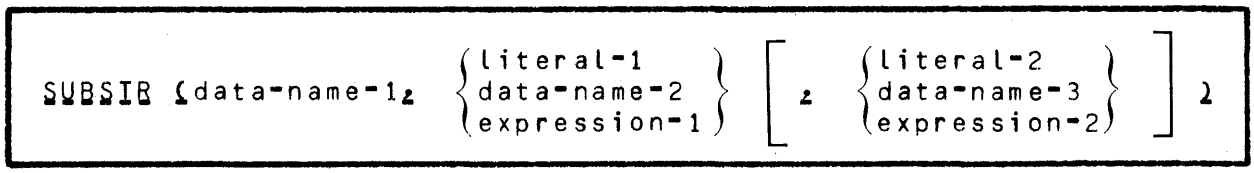
b. Statement (SUBBIT (data-name 1, data-name 2, data-name 3)) is defined as being a pass-by-value because of the extra set of parentheses that surrounds the entire statement.

EXAMPLES.
An example describing the use of the SUBBIT function is as follows:

Examples	Comments
DECLARE SBIT FIXED; SBIT := @(1)00100@; A := SUBBIT (SBIT, 23, 1);	Statement replaces data-name A with a 0.
A := SUBBIT (SBIT, 21, 1);	Statement replaces data-name A with a 1.
DECLARE SBIT BIT (11), AX2 BIT (9); SBIT := @(1) 1101111001@; AX2 := @(1) 100010100@; SUBBIT (AX2, 3) := SUBBIT (SBIT, 3, 2);	Then, AX2 contains 100110000.
DECLARE OBJ.CODE BIT (16), SOC.CODE FIXED; SUBBIT (OBJ.CODE,8,8) := SOC.CODE;	The right-most eight bits of the type FIXED variable SOC.CODE are assigned to the right-most eight positions of OBJ.CODE.

SUBSTR FUNCTION.
The SUBSTR function provides the capability of addressing character substrings within a data-name.

SYNTAX.
The syntactical structure of the SUBSTR function is as follows:



DESCRIPTION.
Data-name 1 is considered of type CHARACTER regardless of the type in its declare statement.

Data-name 2, literal-1, or expression-1 at object time is evaluated as a positive number that is used as the ordinal position of the first character to be accessed within the character-string. The most significant (left-most) character within a character-string is character 0.

Data-name 3, literal-2, or expression-2 is evaluated as a positive number that is used as the number of characters to be accessed within the character-string.

The omitting of data-name 3, literal-2, or expression-2 results in the accessing of the string from the character specified by the value of data-name 2 through the last character in the string.

A range check is performed on data-name 2 and data-name 3, and an out-of-bounds value causes an interrupt of the program. That is, data-name 2 must point into the string, and data-name 3 must not specify more characters than exist between the first character and the end of the string.

A resultant value of 0 (zero) for data-name 3 is valid and results in no accessing of data.

If a SUBSTR function appears to the left of a replacement operator, it is treated as a data-name. Truncation, fill, and data alignment are performed by the operator with type CHARACTER being the destination field-type. That is, if the source field is not of type CHARACTER, the alignment is to the right and is controlled by the data-name 2 position and the number of characters specified by data-name 3. If, however, the source field is of type CHARACTER, the alignment is left-justified to the position as specified by data-name 2 and is controlled by the contents of data-name 3 to determine the number of positions in length.

If data-name 2 or data-name 3 is declared as being data of type CHARACTER, it is evaluated as being a binary number. That is, if a character 1 is given, it is equal to the internal EBCDIC value of 11110001, which converts to a decimal value of 241, which results in the accessing first of the 241st character in the string or a string

length of 241 characters.

The SUBSTR function may be passed to a procedure, by name or by value, according to the following conventions:

a. Statement SUBSTR (data-name 1, data-name 2, data-name 3) is defined as being a pass-by-name.

b. Statement (SUBSTR (data-name 1, data-name 2, data-name 3)) is defined as being a pass-by-value because of the extra set of parentheses that surrounds the entire statement.

EXAMPLES.
In the following examples, assume a data-name of ALPHA that contains a character-string consisting of all 26 letters of the alphabet in sequence from A through Z.

Examples	Comments
`X := SUBSTR(ALFA, 0, 1);`	Data-name X contains an A.
`X := SUBSTR(ALFA, 24);`	Data-name X contains the letters YZ.

`N := 0;`
`DO ODD FOREVER;`
`SUBSTR(PRINT, N, 1) :=`
`SUBSTR(ALFA, 2 * N, 1);`
`N = N + 1;`
`IF (2 * N) GTR 25 THEN`
`UNDO;`
`END ODD;`
	Assume N is type FIXED.
	Data-name PRINT contains every other letter in the string, for example, A C E ... W Y.

`ABC := "OPPOSITE";`
`CH := "VAULT";`
`SUBSTR (ABC, 0, 1) := SUBSTR (CH, 1, 1);`
	This statement replaces data-name ABC with APPOSITE, and CH remains as VAULT.

SWAP FUNCTION.

The SWAP function is used to synchronize asynchronous processes.

SYNTAX.
The SWAP function format is as follows:

```
SWAP (data-name, expression)
```

DESCRIPTION.
The value of the expression is exchanged with the contents of the data-name in one main memory cycle, and the former contents of the data-name are returned by the SWAP function.

The length of the data to be SWAPped is either the length of the data-name or the right-most 24 bits of the data-name, whichever is less. The length of the expression is padded or truncated to the length of the operation in accordance with the rules of the assignment operator.

EXAMPLES.
Examples describing the SWAP function are as follows:

Examples	Comments

`IF SWAP (A,1)`
`  THEN CALL ASSIGN.SPACE;`
`  ELSE CALL< LOOK.FOR.MORE.SPACE;`
	If A contains a 0 (zero), the ELSE portion is executed and A then contains a 1. If A contains a 1, the THEN portion is executed and A then contains a 1.

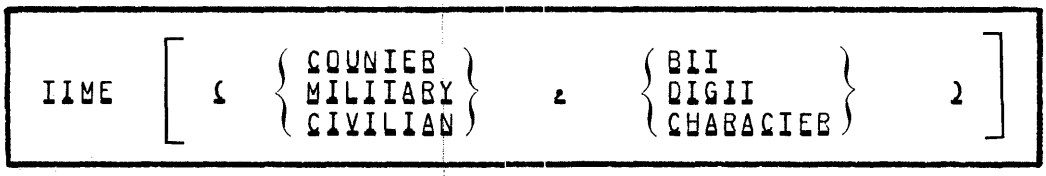
`I: = 3; A(I): = SWAP(I, I+1);`	An equivalent set of statements is as follows:
	`I: = 3;`
	`A(I): =3;`
	`I: = 3 + 1;`

`DECLARE STNG CHARACTER (16),`
`  B CHARACTER (3),`
`  A FIXED;`
`  STNG: = "THE VALUE IS 000";`
`  A: = 123;`
`  B: = SWAP(STNG,CONV`
`  (A,CHARACTER));`
	Data-name B now contains the characters 000 while STNG contains 123. Notice that the right-most 24 bits of the second expression are used regardless of the data-type.

TIME FUNCTION.

The TIME function returns a string that represents the current time of day.

SYNTAX.
The syntactical structure of the TIME function is as follows:



DESCRIPTION.
The TIME function without the option is the same as TIME (CIVILIAN, CHARACTER). The format of each option and string length is as follows:

Option	Format	Bits	Length Digits	Characters
COUNTER	TTTT	20	6	6
MILITARY	HHMMSST	4/6/6/4	2/2/2/1	2/2/2/1
CIVILIAN	HHMMSSTAP	4/6/6/4/16	2/2/2/1/4	2/2/2/1/2

Notation used in the preceding table is as follows:

a. HH equals hours.
b. MM equals minutes.
c. SS equals seconds.
d. T equals 10th of a second
e. AP equals AM (ante meridiem) or PM (post meridiem).

A digit is a 4-bit decimal number.

NOTE
Time durations of less than one 10th of a second may show zero elapsed time.

TODAYS.DATE FUNCTION.
TODAYS.DATE function is a class I reserved word that returns a character-string that represents the time and date the program is compiled.

SYNTAX.
The syntactical structure of the TODAYS.DATE function is as follows:

```
TODAYS.DATE
```

DESCRIPTION.
The date and time are the date and time the program is compiled. The format of the 14-character string that is returned from the TODAYS.DATE function is MM/DD/YY HH:MM.

SECTION 9

HOW TO WRITE A UPL PROGRAM

GENERAL.

The writing of a computer program presupposes an understanding of the problem to be solved and a selection of the programming language most suitable to efficiently solving that problem. Assuming that these conditions are satisfied, the following considerations should be kept in mind as a guide in writing a UPL source language program.

WRITING RULES.

The UPL Compiler accepts a card image input file where columns 1 through 72 may be used for statements, declarations, or comments and where columns 73 through 80 are the card sequence-numbers and/or identification field.

The coding can be specified in a completely free form; that is, any number of statements, declarations, or comments can appear on a single card or over as many cards as desired. Column 72 is considered adjacent to column 1 of the next card. Extra spaces can be used freely throughout the UPL code to improve the readability of the text. A percent sign (%) denotes that the rest of a card is composed of comments. It can be used to delimit the scan procedure, thus increasing compile speed.

EXAMPLES.
For example, the IF statement can be written as:

Example	Comment
IF X EQL Y THEN X := 0; ELSE X := 1;	Each line on the page represents a separate card.

FORM OF A UPL PROGRAM.

Programs are divided into logical units called PROCEDUREs, each having a head statement at its beginning and being terminated with an END statement. PROCEDUREs have an internal structure as described in the procedure statement. A PROCEDURE has a definite ordered relationship to all other PROCEDUREs within a program from either a side-by-side (parallel-PROCEDURE) or subordinate (nested-PROCEDURE) position in that program. The ordering inherently defines the scope or range of a data-name and the PROCEDURE(s) that may be invoked from a given PROCEDURE.

In the description that follows, the main program (lexicographical level 0) is considered a PROCEDURE except that it has no head or END statements and therefore cannot be recursively invoked.

Data-names and nested PROCEDUREs that are used within a PROCEDURE must be declared and completed before any executable statements in that PROCEDURE.

The outer-most PROCEDURE is considered to be the program. The PROCEDURE(s) contained within the program are considered nested at least one level down; that is, they are on lexicographical level 01 or greater, with the maximum depth of 15 sublevels.

Figure 9-1 shows the structure of a typical, though arbitrary, UPL program. Each bracket represents a PROCEDURE and is labeled as being PROCEDURE-n (Psubn) through END-n (Esubn). The declarations and executable statements are indicated as being Dsubn and Xsubn, where n denotes the PROCEDURE to which the statement belongs. Although the number and nesting of PROCEDUREs will vary among programs, the relationship of the parts, declarations, nested-PROCEDUREs, and their executable statements must appear in the order shown. That is, all DECLAREs for a given PROCEDURE must appear in that PROCEDURE before declaration of any nested PROCEDURE and before execution of any statements. When one or more nested PROCEDUREs are declared, however, they must be completed in their entirety (including the executable statements) before the first executable statement of the parent PROCEDURE can be specified.

Five PROCEDUREs, three of which are on lexicographical level 1 (Psub1, Psub2, and Psub3) and two on lexicographical level 2 (Psub4 and Psub5) are shown in figure 9-1. The outer-PROCEDURE is called the program and has no PROCEDURE head or END statements. The FINI card is used to signify the end of compilation.

```
Examples                    Comments

DsubO                       Declare global data-names (lexic level 0).
DsubO
  Psub1                     Begin PROCEDURE 1 (lexic level 1).
    Dsub1                   Local data declarations of PROCEDURE 1.
    Dsub1
    Xsub1                   Executable statements of PROCEDURE 1.
    Xsub1
  Esub1                     END of PROCEDURE 1.

  Psub2                     PROCEDURE 2 (lexic level 2).
    Dsub2                   Local data.
    Dsub2
    .
    .
    .
    Xsub2                   Executable statements of PROCEDURE 2.
    Xsub2
    .
    .
    .
  Esub2                     END of PROCEDURE 2.
  Psub3
    Dsub3                   Local data-names of PROCEDURE 3 that also are
    Dsub3                   global to PROCEDURES 4 and 5 (lexic level 1).
      Psub4                 Local data-names of PROCEDURE 4 (lexic level
        Dsub4               2).
        Dsub4
        .
        .
        .
        Xsub4               Executable statements of PROCEDURE 4.
        Xsub4
      Esub4                 END of PROCEDURE 4.
      Psub5                 Local data-names of PROCEDURE 5 (lexic level
        Dsub5               2).
        .
        .
        .
        Xsub5               Executable statements of PROCEDURE 5.
      Esub5                 END of PROCEDURE 5.
    Xsub3
    Xsub3
    .
    .
    .
  Esub3                     END OF PROCEDURE 3.
XsubO      --------         First executable statement in program.
XsubO
XsubO      --------         Last executable statement in program.
FINI
```

Figure 9-1. Typical UPL Program
Schematic Diagram

Execution of an object UPL Program starts at the first executable statement in the outermost PROCEDURE (statement Xsub0) and is the statement that immediately follows all nested PROCEDUREs. Execution of statements then continues successively from statement to statement within the outermost PROCEDURE or until a STOP statement is encountered.

Since the source code line format in UPL is very flexible, it is suggested that statement levels be indented on new cards to improve the documentation references and the general understanding of a program. Thus, each new PROCEDURE may be indented to a new margin, and its corresponding END may be placed on that same margin. Also, since statements can contain other statements (such as DO, IF, and CASE), each lower statement level may be indented. When a higher level is resumed, its statements should be placed at the proper level margin. It should be noted that this is only a suggestion and that indenting of statements will in no way affect operation of a UPL Program.

Studying the examples and the detailed descriptions of UPL statements and declarations in this manual should aid in understanding how a UPL Program is written.

PROCEDURE CALLING.
Any PROCEDURE can call (invoke) any other PROCEDURE that is currently invoked (any direct ancestor) or any PROCEDURE that is nested one level down within a currently invoked PROCEDURE (any first-generation descendant).

For definitional purposes, the program is considered to be the outermost PROCEDURE and is always in a currently invoked status.

CONCEPT OF SCOPE.
The rule follows directly from the concept of scope. Each PROCEDURE passes all of its declared names as globals to all its descendants. This includes the names of all PROCEDUREs nested one level down. Notice the difference between the name of a PROCEDURE on the current lexic level and the PROCEDURE being named that is on the next lower lexic level.

RELATIONSHIPS.
Let figure 9-2 depict the compile-time relationships of the specified PROCEDUREs.

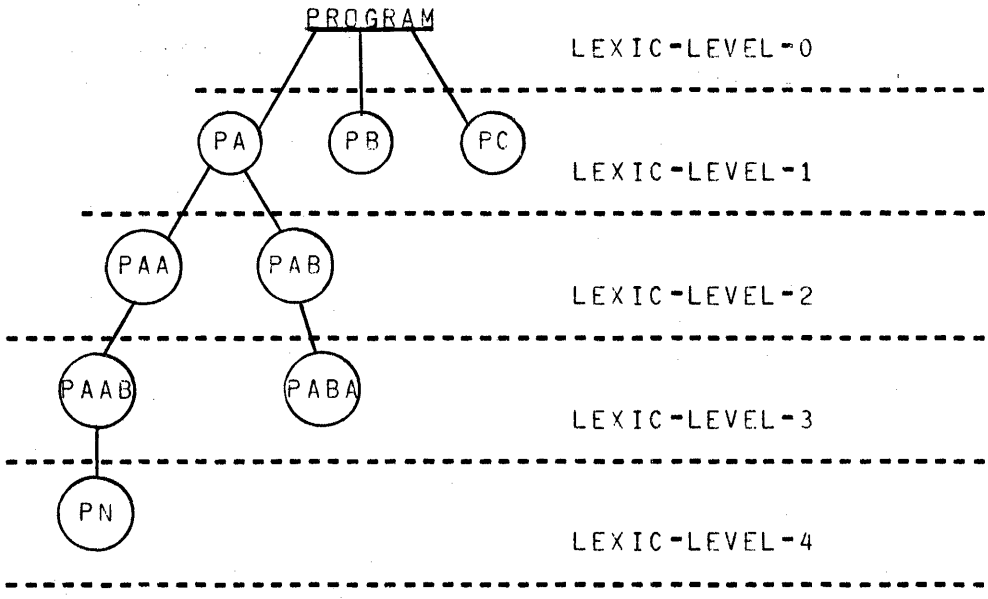


Figure 9-2. Procedure Compile Time Relationships

Then the SCOPE or range of each PROCEDURE is as follows:

a. PROCEDURE PN can invoke any of the following: PN, PAAB, PAA, PA, PAB, PB, or PC.
b. PROCEDURE PB can invoke any of the following: PA, PB, and PC.
c. The parent PROCEDURE can invoke PA, PB, and PC.
d. PROCEDURE PAB can invoke PAB, PABA, PA, PAA, PB, and PC.

As another example, let A, B, C, D, L, M, and K be the names of a set of PROCEDUREs imbedded in some program. If the compile-time relationship of the PROCEDUREs is:

A (B (K), C (L,M), D)

then the SCOPE of a PROCEDURE-invoking statement in each PROCEDURE is:

a. A can call A, B, C, or D.
b. B can call B, K, A, C, or D.
c. K can call K, B, A, C, or D.
d. C can call C, L, M, A, or B.
e. L can call L, C, M, A, B, or D.
f. M can call M, C, L, A, B, or D.
g. D can call D, A, B, or C.

In the previous example, the schematic could be represented as shown in figure 9-3.

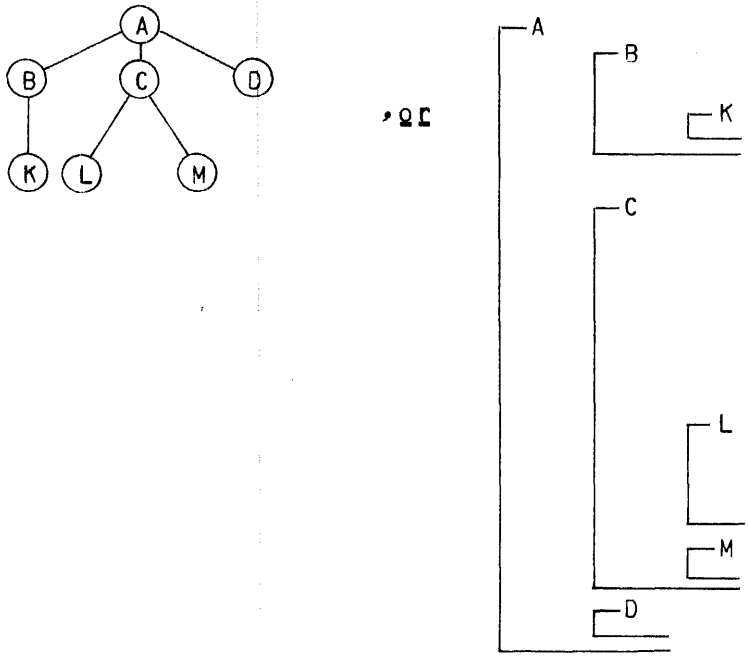


Figure 9-3. Nesting Examples

CODING EXAMPLES.

A flow chart of a program that reads a card, extracts 11 fields of seven columns each, converts each field to a FIXED number, and then prints a copy of each FIXED number is shown in figure 9-4. Two methods that can be used to code this problem follow the flow chart, with the first method (figure 9-5) being a straight-forward approach that follows the flow chart logic closely. The second method (figure 9-6) uses recursive PROCEDURE techniques and more readily exemplifies a typical UPL Program.

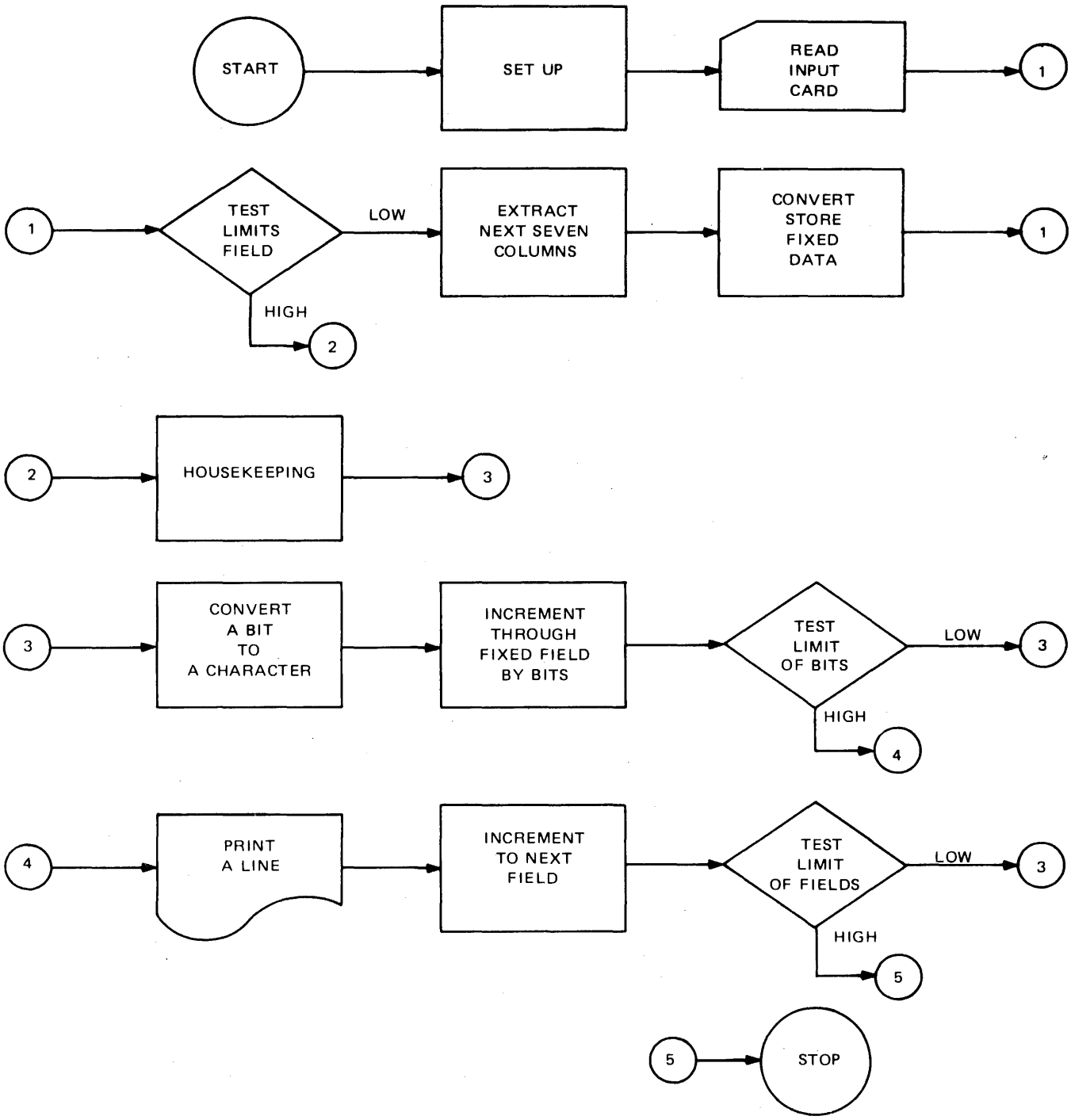


Figure 9-4. Programming Flow Chart

```
BURROUGHS B1700 UPL COMPILER, MARK III.1 (12/20/72 10:21)    TUESDAY,   2/ 6/73,  6:16:51 PM

 LL  NL  SEQUENCE :SOURCE IMAGE                                                    :  PROCEDURE    SEGMENT

  0   0           :$ CONTROL                                                      :

  0   0           :$ SINGLE                                                       :

  0   0           :DECLARE CD CHARACTER (80), CHAR CHARACTER (24), F (11) FIXED ; :
  0   0           :DECLARE ( N,M, COL) FIXED;                                     :
  0   0           :FILE IN ( DEVICE = CARD) ; %                                   :
  0   0           :FILE OUT ( DEVICE = PRINTER ); %                               :
  0   0           :  OPEN IN INPUT;                                               :
  0   0           :  OPEN OUT OUTPUT;                                             :
  0   0           :  N :=M :=COL :=0;                                             :
  0   0           :READ IN (CD);                                                  :
  0   0           :DO PR1 FOREVER;                                                :
  0   1           :     IF COL GTR 70 THEN UNDO PR1;                              :
  0   1           :     F(N) := CONV(SUBSTR(CD,COL,7), FIXED);                    :
  0   1           :     COL := COL + 7;                                           :
  0   1           :     BUMP N;                                                   :
  0   1           :     END PR1;                                                  :
  0   0           :  N:= 0;                                                       :
  0   0           :  DO PR2 FOREVER;                                              :
  0   1           :     M:=0;                                                     :
  0   1           :     DO PR3 FOREVER;                                           :
  0   2           :          SUBSTR(CHAR,M,1) := CONV(SUBBIT(F(N),M,1), CHARACTER,1);  :
  0   2           :          BUMP M;                                              :
  0   2           :          IF M GTR 23 THEN UNDO PR3;                           :
  0   2           :     END PR3;                                                  :
  0   1           :     WRITE OUT (CHAR);                                         :
  0   1           :     BUMP N;                                                   :
  0   1           :     IF N GTR 10 THEN UNDO PR2;                                :
  0   1           :  END PR2;                                                     :
  0   0           :  CLOSE IN;                                                    :
  0   0           :  CLOSE OUT;                                                   :
  0   0           :  STOP;                                                        :
  0   0           :  FINI;                                                        :

***** COMPILATION COMPLETE
```

Figure 9-5. Programming Example 1 (Sheet 1 of 2)

```
COMPILE STATISTICS:

    NUMBER OF ERRORS DETECTED:     0
    NUMBER OF CARDS SCANNED:      30
    NUMBER OF TOKENS SCANNED:     197
    LL ZERO NAME STACK ENTRIES:     8


PROGRAM STATISTICS

    CORE REQUIRED TO RUN:      4864 BITS
    NUMBER OF SEGMENTS:      2
    SIZE OF LARGEST SEGMENT:      1725 BITS
    TOTAL SEGMENT SIZE:       1725 BITS
    DISK SIZE:      8 SEGMENTS


RUN STATISTICS:

    NAME STACK SIZE:     12 ENTRIES
    CONTROL STACK SIZE:    15 ENTRIES
    PROGRAM POINTER STACK SIZE:    25 ENTRIES
    EVALUATION STACK SIZE:    20 ENTRIES
    VALUE STACK SIZE:     1168 BITS
    PROGRAM STATIC CORE:      4736 BITS
    PROGRAM DYNAMIC CORE:         0 BITS


COMPILE TIMES:

    ELAPSED TIME:    0:01:55.2
    PROCESSOR TIME:  (NOT AVAILABLE)
```

Figure 9-5. Programming Example 1 (Sheet 2 of 2)

```
BURROUGHS B1700 UPL COMPILER, MARK III.1 (12/20/72 10:21)    TUESDAY,   2/ 6/73,  6:09:36 PM

 LL  NL  SEQUENCE :SOURCE IMAGE                                                    :  PROCEDURE   SEGMENT

  0   0           :$ CONTROL                                                      :

  0   0           :$ SINGLE DETAIL                                                :

  0   0           :DEFINE CH AS # CHARACTER #, CALL AS ##; % CALL IS A NULL SYMBOL :
  0   0           :DECLARE WORK(11) FIXED, P.NUMB CH (24), CD CH (80);            :


  0   0           :FILE IN (DEVICE = CARD), OUT (DEVICE = PRINTER );              :
  0   0           :  PROCEDURE P1 (X);                                            :
  0   0           :  FORMAL (X) FIXED;                                            :
  1   0           :    IF X LSS 76 THEN CALL P1(X+7);  % A RECURIVE CALL          :  P1

  1   1           :    WORK(X/7-1) := CONV(SUBSTR(CD,X-7,7), FIXED);              :  P1
  1   0           :    RETURN;                                                    :  P1
  1   0           :  END P1;                                                      :  P1
  0   0           :  PROCEDURE P2 (Y);                                            :
  0   0           :    FORMAL (Y) FIXED;                                          :
  1   0           :      PROCEDURE P3 (Z);                                        :  P2
  1   0           :        FORMAL (Z) FIXED;                                      :  P2
  2   0           :        IF Z LSS 23 THEN CALL P3(Z+1);%A RECURSIVE CALL ON P3  :  P3

  2   1           :        SUBSTR(P.NUMB,Z,1) := CONV(SUBBIT(WORK(Y),Z,1),CH,1);  :  P3
CHARACTER
  2   0           :      END P3;                                                  :  P3
  1   0           :    IF Y NEQ 0 THEN CALL P2(Y-1);  %RECURSE P2                 :

  1   1           :    CALL P3(0);                                                :  P2

  1   0           :    WRITE OUT (P.NUMB);                                        :  P2
  1   0           :    END P2;                                                    :  P2
  0   0           :  OPEN IN INPUT;                                               :
  0   0           :OPEN OUT OUTPUT;                                               :
  0   0           :  CALL P1(7);                                                  :

  0   0           :  CALL P2(10);                                                 :

  0   0           :  STOP;                                                        :
  0   0           :  FINI;                                                        :

***** COMPILATION COMPLETE
```

Figure 9-6. Programming Example 2 (Sheet 1 of 2)

```
COMPILE STATISTICS:

    NUMBER OF ERRORS DETECTED:     0
    NUMBER OF CARDS SCANNED:      26
    NUMBER OF TOKENS SCANNED:     211
    LL ZERO NAME STACK ENTRIES:     5


PROGRAM STATISTICS

    CORE REQUIRED TO RUN:      4696 BITS
    NUMBER OF SEGMENTS:     2
    SIZE OF LARGEST SEGMENT:      1409 BITS
    TOTAL SEGMENT SIZE:       1409 BITS
    DISK SIZE:     7 SEGMENTS


RUN STATISTICS:

    NAME STACK SIZE:     10 ENTRIES
    CONTROL STACK SIZE:    15 ENTRIES
    PROGRAM POINTER STACK SIZE:    25 ENTRIES
    EVALUATION STACK SIZE:    20 ENTRIES
    VALUE STACK SIZE:     1096 BITS
    PROGRAM STATIC CORE:      4568 BITS
    PROGRAM DYNAMIC CORE:         0 BITS


COMPILE TIMES:

    ELAPSED TIME:    0:02:06.8
    PROCESSOR TIME:  (NOT AVAILABLE)
```

Figure 9-6. Programming Example 2 (Sheet 2 of 2)

COMMENTS ON PROGRAMMING EXAMPLE 2.
The statement "IF X LSS 76 THEN CALL P1(X+7);" will generate 10 calls to PROCEDURE P1. With each call, an address will point to the next statement that is to be executed when the called PROCEDURE executes a RETURN statement. Each invocation of Psub1 also will generate new space for the new parameter being passed. Run-time statement execution, for example, then will be equivalent to the following sequence of statements:

```
CALL P1 (7);
CALL P1 (14);
CALL P1 (21);
CALL P1 (28);
CALL P1 (35);
CALL P1 (42);
CALL P1 (49);
CALL P1 (56);
CALL P1 (63);
CALL P1 (70);
CALL P1 (77);
 WORK (77/7 - 1 ) := CONV(SUBSTR (CD, 77 - 7, 7), 3);
 WORK (70/7 - 1 ) := CONV(------ (--, 70 - 7, 7), 3);
  .   (63/7 - 1 ) :=  .      .     .   .     .
  .      .         .   .      .     .   .     .
  .      .         .   .      .     .   .     .
 WORK (7/7 - 11)  := CONV(------------, 7 - 7, ---) ---);
```

Procedures P2 and P3 use similar logic counting down by 1 from 10 in P2 for a total of 11 iterations and from 23 to 0 in P3 for 24 iterations. Recursive calls will not generate new code because all procedures in UPL are re-entrant.

SECTION 10

UPL COMPILER CONTROL

COMPILE_DECK.
To ccmpile a UPL Program from cards, the following control cards are required:

```
?COMPILE pg-name[with]UPL      LI[BRARY]

[?FILE STATEMENT CARDS]        FILE cards can be used to relabel the
                               compiler files.  (Refer to FILE
                               statement in the Software Operational
                               Manual.)

?DATA CARDS

[$NEW]

UPL SOURCE CARDS

FINI

?END
```

The UPL Compiler files are:

File	Comment
CARDS	Card source input file.
SOURCE	Primary source file if $MERGE is used.
NEWSOURCE	Updated source output file if $NEW is used.
LINE	Line printer file.

The $NEW compiler control will create a source file on disk that may have other source images merged during compilations.

Example
To compile using a source file on disk and merge additional source images, use the following control cards:

```
?COMPILE      PG-NAME [WITH] UPL LI[BRARY]

[?FILE STATEMENT FOR SOURCE]

[?FILE STATEMENT FOR NEWSOURCE]

?DATA CARDS

$MERGE
```

```
[$NEW]
    UPL SOURCE IMAGES TO BE MERGED (PATCHED)
    .
    .
    .
    UPL SOURCE IMAGE WITH SEQUENCE FIELD EQUAL 99999999

FINI

?END
```

COMPILER CONTROL CARD OPTIONS.

All compiler control cards must have a $ (dollar sign) in column 1. Control options may appear anywhere from columns 2 through 71 and must be separated by a space. Columns 72 through 80 are for sequence numbers.

The word "NO" may appear before most options. It turns off or reverses the effect of the option.

The following is an alphabetical listing of the options and their actions.

Options	Actions
AMPERSAND	Prints those ampersand cards that are examined.
CHECK	Checks the source input file for sequence errors.
CODE	Prints the UPL-object code generated for each source statement.
CONTROL	Turns on the printing for all following control cards. To see the control option printed requires two control cards.
CSSIZE INTEGER	Overrides the compiler estimate of the control stack size. Integer is in number of entries.
DETAIL	Prints the expansion of all define invocations.
DOUBLE	Double spaces the listing.
DYNAMICSIZE INTEGER	Overrides the compiler estimate of the memory allocated for paged arrays. INTEGER is the number of

Options	Actions
	bits allocated.
ESSIZE INTEGER	Overrides the compiler estimate for the size of the evaluation stack. INTEGER is the number of entries.
FORMAL.CHECK	The actual parameters passed to each procedure will be checked, at execution time, against the types and length specifications of their corresponding formal declarations. Also, the values returned from function procedures will be checked against the type and length in the procedure head statement. Lack of correspondence is a run-time error.
INTERPRETER INTERPRETER MULTI-FILE-NAME/INTERPRETER/FILE NAME	Changes the default interpreter ID from UPL/INTERP1. When the program (being compiled) is executed, it will require a new interpreter as specified.
INTRINSIC INTRINSIC-FAMILY-NAME	Changes the family name of the intrinsics to be used when the program (being compiled) is executed. The default intrinsic family name is UPL.INTRIN.
LIST	Prints the source input that was compiled. A NO list also will turn off the LISTALL options.
LISTALL	Prints all source input whether or not conditionally excluded. The LISTALL turns on LIST, but NO LISTALL does not turn off LIST.
MERGE	The primary source file is on tape or disk and will have cards merged from the card reader.
NEW	Creates a new primary source file.
NO	Turns off or reverses the effect of any option that immediately follows it.

Options	Actions
NSSIZE INTEGER	Overrides the compiler estimate of the name stack size. INTEGER is in number of names.
PAGE	Causes a skip to the top of a new page listing.
PPSIZE INTEGER	Overrides the compiler estimate of program pointer stack size. INTEGER is the number of entries in the stack.
SEQ (BEGINNING-NUMBER INCREMENT)	Resequences the new primary output file.
SINGLE	Single spaces the listing.
SIZE	Prints code segment names and sizes at the end of the compile.
SUPPRESS	Suppresses warning messages. To suppress sequence error messages, use NO CHECK.
VOID SEQUENCE NUMBER	Voids or removes records in the primary source file. Begins at the sequence number of the VOID card and goes through the sequence number following the word VOID. The VOID card may not be preceded by a NO, must be the only compiler option on the card, and must contain sequence numbers in columns 72 through 80.
VSSIZE INTEGER	Overrides the compiler estimate for the value stack size. INTEGER is in bits.
XMAP	Creates an extended UPL-object code MAP file showing the relative displacement of object code per source card sequence number per code segment.

APPENDIX A

CLASS I RESERVED WORDS

ACCEPT	AND	AS
BASE	BIT	BUMP
BY		
CASE	CAT	CHANGE
CHARACTER	CLEAR	CLOSE
DECLARE	DECREMENT	DEFINE
DISPLAY	DO	DUMMY
DYNAMIC		
ELSE	END	EQL
EXOR		
FILE	FILLER	FINI
FIXED	FORMAL	FORMAL.VALUE
FORWARD	FROM	
GEQ	GTR	
IF	INTRINSIC	
LEQ	LSS	
MOD		
NEQ	NOT	
OF	OPEN	OR
PAGED	PROCEDURE	
READ	RECEIVE	REMAPS
SEEK	SEGMENT	SEND
SKIP	SPACE	STOP
SUBBIT	SUBSTR	
THEN	TO	TODAYS.DATE
UNDO	USE	
VARYING		
WRITE		

APPENDIX B

CLASS II RESERVED WORDS

BASE.REGISTER
BINARY
CONV
CONVERT
DATE
DECIMAL
LENGTH
LIMIT.REGISTER
MAKE.READ.ONLY
M.MEM.SIZE
NAME.OF.DAY
REVERSE.STORE
SEARCH.LINKED.LIST
S.MEM.SIZE
SWAP
TIME

APPENDIX C

CLASS III RESERVED WORDS

ASCII	AREAS
BACKUP	CARD
CRUNCH	CRUNCHED
DEVICE	DISK
DISK.FILE	DISK.FILE.1
DISK.FILE.2	DISK.PACK
DISK.PACK.CAELUS	DISK.PACK.CENTURY
EBCDIC	EOF
END.OF.TEXT	EVEN
FORMS	INPUT
LABEL	LOCK
LOCK.OUT	MULTI.FUNCTION.CARD
NEW	NO.REWIND
ODD	ON
OUTPUT	
PAPER.TAPE.PUNCH	PAPER.TAPE.READER
PARITY	PRINTER
PUNCH	PURGE
RANDOM	RECORDS
REEL	RELEASE
REMOVE	SECURITY.ID
SERIAL	SORTER.READER
SPO	TAPE
TAPE.7	TAPE.7.CLUSTER
TAPE.7.UPRIGHT	TAPE.9
TAPE.9.CLUSTER	TAPE.9.NRZ
TAPE.9.PE	UNIT
VARIABLE	WITH