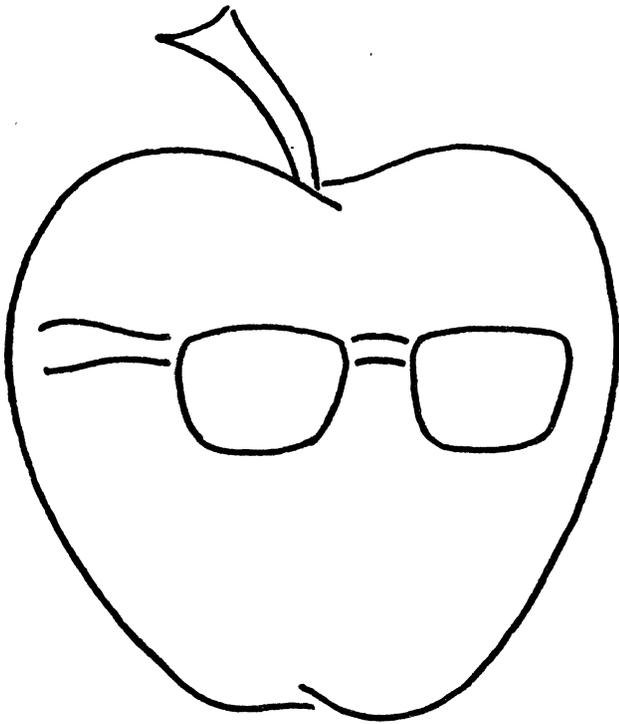


March 31, 1972



Apple

Reference Manual

by

Fred Krull

Michael Marcotty

Mary Pickett

James Thomas

Ronald Zeilinger

Computer Science Department

 **Research Laboratories**
General Motors Corporation

31 MARCH 1972

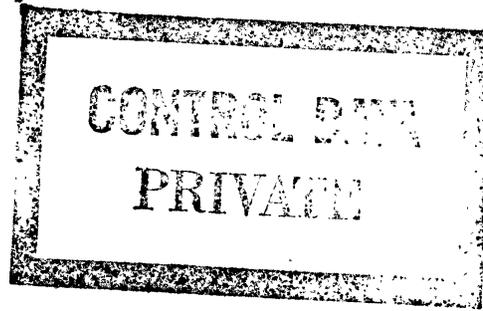


TABLE OF CONTENTS

PREFACE	9
CHAPTER 1: PROGRAM ELEMENTS	10
INTRODUCTION	10
LANGUAGE CHARACTER SET	10
Collating Sequence	12
Length of Identifiers	12
Keywords	12
Statement Identifiers	13
Attribute Keywords	13
Built-in Function Names	13
Option Keywords	13
Conditions	13
DELIMITERS	14
Arithmetic Operators	14
Relational Operators	15
Bit-string Operators	15
String Operator	15
Parentheses	15
Separators and Other Delimiters	16
COMMENTS	16
The Use of Blanks and Comments	16
ELEMENTARY PROGRAM STRUCTURE	17
Simple Statements	17
Compound Statements	17
Prefixes	17
CHAPTER 2: PROGRAM STRUCTURE	19
INTRODUCTION	19
STATIC PROGRAM STRUCTURE	19
Groups	19
Block Structure	20
Use of the END Statement	22
DYNAMIC PROGRAM STRUCTURE	22
Procedure References	23
Subroutine References	24
Function References	24
Activation and Termination of Blocks	24
The Environment of a Block	26

ARGUMENT PASSING	27
Parameters	28
Correspondence of Arguments and Parameters	29
Use of Dummy Arguments	30
Entry References as Arguments	31
Use of the ENTRY Attribute	32
CHAPTER 3: DATA ELEMENTS	33
INTRODUCTION	33
DATA TYPES	33
PROBLEM DATA	33
Arithmetic Data	34
Scale	34
Precision	34
Arithmetic Constants	36
String Data	36
Character-String Data	37
Character-String Constants	37
Bit-String Data	38
Bit-String Constants	38
PROGRAM-CONTROL DATA	38
Label Data	38
Statement-Label Constants	38
Statement-Label Variables	39
Locator Data	39
Locator Qualification	40
Interrupt Data	41
File Variable	41
Entry Data	42
ORGANIZATION	43
Scalar Items	43
Scalar Variables	43
Data Aggregates	43
Arrays	43
Structures	44
Arrays of Structures	45
Attributes of Structures	46
NAMING	46
Simple Names	46
Subscripted Names	46
Qualified Names and Ambiguous References	47
Subscripted Qualified Names	50
CHAPTER 4: DATA MANIPULATION	52
INTRODUCTION	52

31 MARCH 1972

EXPRESSIONS	52
Arithmetic operations	54
Descriptor Arithmetic	55
Relational Operations	56
Bit-string Operations	57
String Operations	58
EVALUATION OF EXPRESSIONS	59
Priority of Operators	59
Use of Parentheses	60
Example of Expression Evaluation	60
ARRAY EXPRESSIONS	63
Operations between Arrays and Scalars	63
Operations between Arrays	64
DATA CONVERSION	65
ARITHMETIC CONVERSION	65
Results of Arithmetic Operations	66
TYPE CONVERSION	67
1. Arithmetic Conversion	68
2. Character-string to Arithmetic	68
3. Conversions to Character-string	69
4. Bit-string to Arithmetic	69
5. Arithmetic to Bit-string	69
6. Offset to Pointer	69
7. Pointer to Offset	70
8. Descriptor to Pointer	70
9. Pointer to Descriptor	70
10. Offset to Descriptor	70
11. Descriptor to Offset	70
12. Arithmetic to Locator	71
13. Character-string to Entry Value	71
CHAPTER 5 -- DATA DESCRIPTION	72
INTRODUCTION	72
DECLARATIONS	72
EXPLICIT DECLARATIONS	73
Label Prefixes	74
Parameters	74
CONTEXTUAL DECLARATIONS	74
SCOPE OF DECLARATIONS	75
DEFAULT ATTRIBUTES	79
LIST OF ATTRIBUTES	80
AUTOMATIC, STATIC, REGISTER, and BASED	81
BINARY and DECIMAL	88
BIT and CHARACTER	89
BUILTIN	90

CHARACTER	91
CONSTANT	91
CONDITION and EVENT	92
DECIMAL	92
DESCRIPTOR	92
Dimension	93
ENTITY	94
ENTRY	95
EVENT	96
EXTERNAL and INTERNAL	97
FILE	97
FILE_SET	98
FIXED and FLOAT	99
INITIAL	99
LABEL	102
LIKE	103
OFFSET, POINTER, and DESCRIPTOR	104
REFER	105
RETURNS	106
SET	107
VARIABLE	108
VARYING	108
CHAPTER 6: FILE HANDLING	109
INTRODUCTION	109
FILES	109
Sequential Files	110
Structured Files	111
File Variables	111
SEQUENTIAL FILE HANDLING	113
Use of GET and PUT Statements	113
Data Specification	114
Data Lists	115
Format Lists	116
Data Format-Items	118
Control Format-Items	125
STRUCTURED FILE HANDLING	127
Storage Management	127
Entities	128
Sets	129
Creating and Deleting Associations	131
Searching a Set	132
Associative Data Built-In Functions	133
CHAPTER 7 - INTERRUPT HANDLING	134

31 MARCH 1972

INTRODUCTION	134
CONDITIONS	134
System Conditions	134
Programmer-Defined Conditions	135
EVENTS	135
Event Declarations	135
Event States	136
Completion State	136
Delay State	137
Use of the ONPTR Built-in Function	137
USE OF INTERRUPT-HANDLING STATEMENTS	138
Use of the ON Statement	138
Use of the REVERT Statement	141
Use of the SIGNAL Statement	142
Use of the WAIT Statement	142
Use of the LOCK and UNLOCK Statements	143
CHAPTER 8: STATEMENTS	144
INTRODUCTION	144
CLASSIFICATION OF STATEMENTS	144
The ALLOCATE Statement	145
The Assignment Statement	150
The BEGIN statement	154
The CALL statement	154
The CREATE statement	156
The DECLARE statement	156
The DELETE statement	159
The DO statement	159
The END statement	164
The ENTRY statement	164
The EXIT statement	165
The FIND Statement	166
The FOR EACH statement	169
The FREE statement	171
The GET Statement	173
The GO TO statement	173
The IF statement	175
The INSERT statement	177
The LET statement	178
The LOCK statement	179
The null statement	180
The ON statement	180
The PROCEDURE statement	182
The PUT statement	183
The REMOVE statement	183

The RETURN statement185
 The REVERT statement185
 The SIGNAL statement186
 The UNLOCK statement187
 The WAIT statement188

APPENDIX 1 - BUILT-IN FUNCTIONS, PROCEDURES, AND

PSEUDO-VARIABLES190
 INTRODUCTION190
 ARITHMETIC FUNCTIONS192
 ABS (x)192
 CEIL (x)192
 FLOOR (x)192
 MOD (x, d)192
 ARRAY FUNCTIONS192
 DIM (a, d)192
 HBOUND (a, d)192
 LBOUND (a, d)193
 ASSOCIATIVE DATA FUNCTIONS193
 ALL193
 APLESET (s)193
 APLEVAR (s)193
 APLINDX (e, s, c)194
 APLNUMB (s, c)194
 APLOWNI (e, s, c)194
 APLOWRS (e, c, d)194
 APLSNAM (e, i, j)195
 APLTYPE (e)195
 CONVERSION FUNCTIONS195
 BYTE (x[, i])195
 CHAR (v[, l])196
 ENTRY (c)196
 FIXED (x[, p])197
 FLOAT (x[, p])197
 HEX (f[, i[, l]])197
 OFFSET (p, f)198
 POINTER (o, f)198
 INTERRUPT HANDLING FUNCTIONS198
 COMPLETION (e)198
 DELAY (e)198
 ONFILE198
 ONLOC199
 ONPTR (e)199
 MATHEMATICAL FUNCTIONS199
 ATAN (x)199

31 MARCH 1972

COS (x)199
LOG (x)199
SIN (x)199
SQRT (x)200
TAN (x)200
STORAGE MANAGEMENT FUNCTIONS200
ADDR (v)200
DESCR (l, a)200
FILE (g)200
NULL200
STRING HANDLING FUNCTIONS201
INDEX (s, p)201
LENGTH (s)201
RAL (b)201
SUBSTR (s, i[, j])201
MISCELLANEOUS FUNCTIONS202
DATE202
INLINE (f, r, s, t), INLINE (f, g, x, a, y, b, z, c)202
TIME204
APPENDIX 2 - CONDITIONS205
INTRODUCTION205
CONVERSION Condition205
ENDFILE Condition206
ERROR Condition206
FIND Condition206
FINISH Condition207
OVERFLOW Condition207
Programmer-defined Condition207
STORAGE Condition208
UNDEFINEDFILE Condition208
UNDERFLOW Condition208
ZERODIVIDE Condition209
APPENDIX 3 - KEYWORDS, ABBREVIATIONS AND SYNONYMS210
KEYWORD ABBREVIATIONS210
KEYWORD SYNONYMS213
APPENDIX 4 - DATA CHARACTER SET214
APPENDIX 5 - COMPILE-TIME CONTROLS217
APPENDIX 6 - NOTATION219
APPENDIX 7 - STRUCTURE MAPPING221
APPENDIX 8 -- LITERALLY223

31 MARCH 1972

PREFACE

This manual serves as a reference to the Apple Programming Language as implemented for the STAR computer system. The Apple Language itself is a dialect of PL/I; that is, Apple is a superset of a subset of PL/I.

In drafting the specifications for Apple, the rules of PL/I have been closely followed. The deviations from PL/I have been in the main to disallow certain operations, statements, data types, etc. The rules of precision have been changed to take into account the architecture of the STAR computer. Thus, as long as a program was written within the defined subset of PL/I, it should compile correctly.

The supersetting of the language has been to provide support for systems programming and to integrate the APL (Associative Programming Language) statements directly into the language. Programmers may declare and reference the new storage class REGISTER and cause any STAR machine instruction to be emitted through use of the INLINE built-in procedure. The associative data manipulation statements FIND, FOR EACH, INSERT, REMOVE, and LET have been added to the language (CREATE and DELETE are synonymous with ALLOCATE and FREE). These statements may be used to manipulate two new data constructs, ENTITY and SET.

The programmer who is preparing to use Apple should give careful attention to the specifications contained in this manual. Particular attention should be given to Chapters 3, 4, and 5, where the rules differ considerably from PL/I.

CHAPTER 1: PROGRAM ELEMENTSINTRODUCTION

An Apple program can be regarded simply as a string of characters. Chapter 1 defines the elements of the language in terms of character elements and describes special significance that has been assigned to particular characters or combinations of characters.

LANGUAGE CHARACTER SET

The Apple language is based on a 60-character set. The character set is composed of alphabetic characters, digits, and special characters. There are 29 alphabetic characters, the letters A through Z and three additional characters that are defined as and treated as alphabetic characters. These characters and the graphics by which they are represented are:

<u>Name</u>	<u>Graphic</u>
Number symbol	#
At symbol	@
Dollar symbol	\$

There are ten digits. Decimal digits are the digits 0 through 9. A binary digit (bit) is either a 0 or a 1. The hexadecimal digits include the ten decimal digits and the alphabetic characters A through F. An alphameric character is either an alphabetic character or a decimal digit. There are 21 special characters. These characters and the gra-

31 MARCH 1972

phics by which they are represented in this manual are:

<u>Name</u>	<u>Graphic</u>
Blank	␣
Equal or Assignment symbol	=
Plus	+
Minus	-
Asterisk or Multiply symbol	*
Slash or Divide symbol	/
Left Parenthesis	(
Right Parenthesis)
Comma	,
Decimal Point or Period	.
Single quotation mark	'
Double quotation mark	"
Semicolon	;
Colon	:
Not symbol	~
Or symbol	!
And symbol	&
Greater-than symbol	>
Less-than symbol	<
Break character	-
Percent symbol	%

31 MARCH 1972

Some keywords may be written in an abbreviated form; these are listed in Appendix 3.

Statement Identifiers

A statement identifier is a sequence of one or more keywords used to define the function of a statement (see "Simple Statements" below).

Examples:

```
GO TO
DECLARE
ALLOCATE
```

Attribute Keywords

Attribute keywords are used for the specification of some attributes.

Examples:

```
FLOAT
CHARACTER
```

Built-in Function Names

A built-in function name is a keyword that is the name of an algorithm provided by the language and accessible to the programmer (see "Function References" in Chapter 2).

Examples:

```
LENGTH
DATE
```

Option Keywords

An option keyword is used to influence the execution of a statement.

Examples:

```
SET
REMOTE
```

Conditions

A condition is a keyword used in the ON, SIGNAL, and REVERT statements. The programmer may specify special action on

31 MARCH 1972

occurrence of a condition (see Chapter 7).

Examples:

OVERFLOW
ZERODIVIDE

DELIMITERS

Certain single characters and certain combinations of characters are used as delimiters and fall into six classes:

arithmetic operators
relational operators
bit-string operators
string operators
parentheses
separators and other delimiters

Arithmetic Operators

The arithmetic operators are:

+ denoting addition or prefix plus
- denoting subtraction or prefix minus
* denoting multiplication
/ denoting division
** denoting exponentiation

31 MARCH 1972

Relational Operators

The relational operators are:

- > denoting greater than
- > denoting not greater than
- >= denoting greater than or equal to
- = denoting equal to
- = denoting not equal to
- <= denoting less than or equal to
- < denoting less than
- < denoting not less than

Bit-string Operators

The bit-string operators are:

- ~ denoting not
- & denoting and
- ! denoting or

String Operator

The string-operator is:

- !! denoting concatenation

Parentheses

Parentheses are used in expressions, for enclosing lists, and for specifying information associated with various keywords.

- (left parenthesis
-) right parenthesis

31 MARCH 1972

Separators and Other Delimiters

<u>Name</u>	<u>Graphic</u>	<u>Use</u>
comma	,	separates elements of a list
semicolon	;	terminates statements
assignment	=	used in assignment, DO, FIND, LET, and FOR EACH statements
colon	:	used in label prefixes and in bound specifications
blank		used as a separator
period	.	separates items in qualified names
arrow	->	qualifies a reference to a based variable
percent	%	designates compiler control statements (see Appendix 5 for description)

COMMENTS

General format:

```
comment ::= /* comment-string */
```

where "comment-string" contains any of the characters of the language character set except the combination "*/".

Comments are used for documentation only and do not participate in the execution of a program.

The Use of Blanks and Comments

Identifiers, constants (except character-string constants), and composite operators (e.g., !!) may not contain blanks. Identifiers, constants, and keywords may not be immediately adjacent. They must be separated by an operator, assignment symbol, arrow, parenthesis, colon, semicolon, comma, period, blank, or comment. Additional intervening blanks or comments are always permitted. Blanks are optional between keywords of the statement identifiers GO TO and FOR EACH.

31 MARCH 1972

ELEMENTARY PROGRAM STRUCTURE

An Apple program is constructed from basic program elements called statements. There are two types of statements, simple and compound. Statements are grouped into larger program elements, the group and the block. These are discussed in Chapter 2.

Simple Statements

General format:

```
simple-statement ::= [[statement-identifier] statement-body];
```

The "statement-identifier", if it appears as a keyword, characterizes the kind of statement. If it does not appear, and the "statement-body" appears, then the statement is an assignment statement. If only the semicolon appears, the statement is a null statement.

Compound Statements

A compound statement is a statement that contains other program elements. There are two types of compound statements:

The IF compound statement

The ON compound statement

The final statement contained in a compound statement is a simple statement and thus has a terminal semicolon. Hence the compound statement will automatically be terminated by this semicolon.

Each Apple statement is described in the alphabetic list of statements in Chapter 8.

Prefixes

Statements may be labeled to permit reference to them through the use of label prefixes.

General format:

```
label-prefix ::= identifier :
```

Label-prefix identifiers are called labels and may be used to refer to the statement that they prefix. Labels appearing before PROCEDURE and ENTRY statements are special cases

31 MARCH 1972

and are known as entry names (see "Procedure References" in Chapter 2). All other labels are called statement labels. A name appearing before a statement is said to be explicitly declared with the attribute of a label constant by virtue of its appearance as a label prefix. Only one label prefix may precede a single statement, and the label prefix may not be subscripted.

31 MARCH 1972

CHAPTER 2: PROGRAM STRUCTUREINTRODUCTION

A program is composed of one or more separately compiled procedures. At execute time, those procedures that are required to solve a particular problem are dynamically (i.e., at first reference) linked together. Thus, the collection of procedures used to solve any problem may be data dependent and may vary from one execution to the next.

This chapter describes the following:

1. The static structure of a program as specified at compile time.
2. The dynamic structure of a program that is established at execute time.
3. The rules by which data may be passed between procedures at execute time.

STATIC PROGRAM STRUCTURE

A procedure is made up of basic elements called statements. A statement may be either a simple statement or a compound statement. Statements may be collected together at compile time into larger units, called groups and blocks.

Groups

A group is a collection of one or more statements that may be considered as a single statement for the purposes of control.

General format:

```
group ::=
```

```
    [label:] group-statement [statement]... END [label];
```

```
group-statement ::= do-statement | for-each-statement
```

The label following the END is the label of the group-statement (see "Use of the END statement" in this chapter).

31 MARCH 1972

The group-statement may specify iteration or selection (see "The DO statement" and "The FOR EACH statement" in Chapter 8).

Each "statement" in the body of the group may be a simple-statement, compound-statement, group, or begin-block.

Block Structure

A block is a collection of statements that defines the program region (or scope) throughout which an identifier is established as a name with an associated set of attributes. A block is also used for control purposes.

There are two kinds of blocks, begin blocks and procedure blocks.

General format:

begin-block ::=

```
[label:] begin-statement [statement]... END [label];
```

procedure-block ::=

```
label: procedure-statement [statement]... END [label];
```

Each "statement" in the body of a begin-block or procedure-block may be a simple-statement, compound-statement, group, begin-block, or procedure-block.

The label following END is the label of the corresponding BEGIN statement or PROCEDURE statement. While the label of the BEGIN statement is optional, the PROCEDURE statement must have a label. The label required for the PROCEDURE statement serves as the procedure name. The procedure name gives a means of activating the procedure at its primary entry point. Secondary entry points can also be defined for a procedure by the use of the ENTRY statement.

Although the begin block and the procedure have a physical resemblance and play the same role in delimiting scope of names (see "Scope of Declarations" in Chapter 5) and defining allocation and freeing of storage (see "Activation and Termination of Blocks" in this chapter), they differ in an important functional respect. A begin block, like a single statement, is activated by normal sequential flow (except when used as an on-unit), and it can appear wherever a single statement can appear. A procedure can only be activated remotely by CALL statements or by function

31 MARCH 1972

references. When a program containing a procedure is executed, control passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of the procedure.

As the above definition of block implies, any block A can include another block B, but partial overlap is not possible. Block B must be completely included in block A. Such nesting may be specified to any depth. A procedure that is not included in any other block is called an external procedure. A procedure included in some other block is called an internal procedure. Every begin block must be included in some other block. Hence, the only external blocks are external procedures. All of the text of a begin block except the label of the BEGIN statement of the block is said to be contained in the block. All of the text of a procedure except the entry names of the procedure is said to be contained in the procedure. That part of the text of a block B that is contained in block B, but not contained in any other block contained in B, is said to be internal to block B. The entry names of an external procedure are not internal to any procedure and are called external names.

Example:

```

A:
PROCEDURE:
  statement-1
  B:
  BEGIN;
    statement-2
    statement-3
  END B;
  statement-4
  C:
  PROCEDURE:
    statement-5
  X:
  ENTRY;
    D:
    BEGIN;
      statement-6
      statement-7
    END D;
    statement-8
  END C;
  statement-9
END A;

```

31 MARCH 1972

In this example, statements 1 through 9 are labeled or unlabeled simple or compound statements or groups. As the brackets on the right indicate, block A contains blocks B and C, and block C contains block D. Block A is an external procedure. The procedure name is A, which is an external name and is the only entry name for the procedure. X is an entry name corresponding to a secondary entry point for procedure C. Blocks B and D are begin blocks. Block C is an internal procedure.

Use of the END Statement

The END statement may contain an optional label. If the optional label following END is not used, the END statement terminates that unterminated group or block headed by the DO, FOR EACH, BEGIN, or PROCEDURE statement that physically precedes, and appears closest to, the END statement. If a label is used following an END statement, the action is exactly the same except that a check is made that the statement at the head of the block or group being terminated is labeled with the same label as is specified with the END statement. If a match is not found, an error message is generated.

DYNAMIC PROGRAM STRUCTURE

A begin block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when the procedure is invoked at any one of its entry points. A block may be active during certain time intervals of the execution of a program. A block is active if it has been activated and is not yet terminated. A procedure-block may be either an internal procedure or an external procedure. Internal procedure references are resolved at compile time, while external procedure references are resolved at execute time. If an internal procedure is referenced, it must be internal to a block that is active at the time of invocation.

Each procedure invocation implies the activation of a new block that is a descendent of a previous block. However, the order or sequence of invocation is a function of the problem and may dynamically change from one execution to the next. At the invocation of a new block, generations of data items may be created. These data items may be referenced in descendent blocks subject to the rules of scope as described in Chapter 5. Data items declared with the STATIC attribute will be allocated and initialized once at the time the first block in which they are declared is activated.

31 MARCH 1972

Procedure References

At any point in a program where an entry point of a given procedure is known, either directly through its name or indirectly through the use of an entry variable, and the procedure is internal to an active block, the procedure may be invoked. A reference to a procedure has the form:

entry-expression [(argument [, argument]...)]

where "entry-expression" may be:

1. an entry constant
2. an entry variable

Each entry constant or variable must be declared, either through its appearance as a label prefix in a PROCEDURE or ENTRY statement or through the use of the ENTRY attribute in a DECLARE statement (see "ENTRY" in Chapter 5). Either declaration indicates the number (possibly zero) and data types of the parameters for the procedure. The number and data types of the arguments in the procedure reference must match the number and data types of the parameters indicated in the declaration. The matching is checked at compile time. When a procedure reference invokes a procedure, each argument specified in the reference is associated with its corresponding parameter in the list for the denoted entry point, and control is passed to the procedure at the referenced entry point.

There are two distinct uses of procedures, determined by one of two contexts in which a procedure reference may appear:

1. A procedure reference may appear following the keyword CALL in a CALL statement. In this case, the procedure is invoked as a subroutine procedure, or simply a subroutine.
2. A procedure reference may appear as an operand in an expression. In this case, the reference is said to be a function reference, and the procedure is invoked as a function.

Any procedure may be invoked as either a function or a subroutine. However, the RETURN statement in a procedure invoked as a function must specify a return value. If a procedure is invoked as a subroutine, any value given in a RETURN statement is ignored. (See "The RETURN Statement" in Chapter 8.)

31 MARCH 1972

Subroutine References

A subroutine reference transfers control to an entry point of a procedure and activates the procedure. Activation of the subroutine may be terminated by execution of a RETURN statement or by the END statement of the block.

A value is not returned by a subroutine, but values obtained in a subroutine may be made known in the invoking procedure either by assigning a value to a variable known in the invoking procedure or by assigning a value to a parameter which has not been passed as a dummy argument.

Function References

When a function reference appears in an expression, the procedure is invoked. The result of the execution of the procedure is the value of the function, which is passed (with the return of control) back to the point of invocation. This returned value is then used to evaluate the expression.

The procedure invoked by a function reference normally will terminate execution with a statement of the form:

```
RETURN (expression);
```

It is the value of this expression that will be returned as the function value.

Besides function references to procedures written by the programmer, a function reference may invoke one of a set of built-in functions. The set of built-in functions is an intrinsic part of Apple. It includes commonly used arithmetic functions, functions for manipulating strings and arrays, and other functions related to special facilities provided in the language. The identifiers corresponding to the built-in function names are not reserved; any such identifier can be used by the programmer for other purposes subject to the rules of scope (see Chapter 5). The complete list of these functions and their descriptions can be found in Appendix 1.

Activation and Termination of Blocks

Blocks can be activated in a variety of ways. A begin block is activated by normal sequential flow of control. In all cases, a begin block must be contained within an active procedure block at the time of activation.

31 MARCH 1972

Procedure blocks, on the other hand, can only be activated by CALL statements or by function references. When a procedure containing internal procedures is executed, control will pass around each internal procedure from the PROCEDURE statement to the corresponding END statement.

There are a number of ways in which a block may be terminated. A begin block is terminated when control passes through the END statement for the block. A procedure block is terminated on execution of a RETURN statement or an END statement for the block. (In this case the END statement implies a RETURN statement.) A block is terminated on execution of a GO TO statement contained in the block that transfers control to a point not contained in the block. Any intervening blocks are also terminated.

If a block B is activated and control stays at points internal to B until B is terminated, no other blocks can have been activated while B was active. However, another block, B1, may be activated from a point internal to block B while B still remains active. This is possible only in the following cases:

1. B1 is a procedure block immediately contained in B (i.e., the label of B1 is internal to B) and reached through a procedure reference.
2. B1 is a begin block internal to B and reached through normal flow of control.
3. B1 is a procedure block not contained in B and reached through a procedure reference. (B1, in this case, may be identical to B, i.e., a recursive call. However, it is to be regarded as a dynamically different block).
4. B1 is a begin block or a statement specified by an ON statement (see "The ON Statement" in Chapter 8) and reached because of an interrupt. (For present purposes, even if B1 is a statement, it can be regarded as a block; this case is dynamically similar to case 1 or case 3 above.)

In any of the above cases, while B1 is active, it is said to be an immediate dynamic descendant of B. Block B1 may itself have an immediate dynamic descendant B2, etc., so that a chain of blocks (B, B1, B2, ...) is created, where, by definition, all of the blocks are active. In this chain, each of the blocks B1, B2, etc., is said to be a dynamic descendant of B. When a block B is terminated, all of the

31 MARCH 1972

dynamic descendants of B are also terminated. Storage for all automatic variables declared in these blocks will be released at the time of termination. If a block B1 is a dynamic descendant of a block B, then block B dynamically encompasses block B1.

The Environment of a Block

On activating a block, certain initial actions are performed, e.g., allocation of storage for automatic variables. These initial actions constitute the prologue. After the prologue has executed, the following are available for computation:

1. Established generations of automatic and register variables declared outside the block and known within it.
2. Static variables known within the block, and register and automatic variables declared in the block.
3. Arguments passed to the block.

When several activations of B are in existence, as in recursion, it is essential to know the activation of B that holds the storage of data declared in B and known to descendant blocks. If a block B1 is statically nested within n containing blocks, the particular activation of each of the n blocks that hold the generations of data known to B1 form the environment of the activation of B1.

When an entry name is assigned to an entry variable, the environment to be used in subsequent invocations is determined and forms part of the entry value. This environment is the activation of the block that contains the procedure whose entry name is assigned. The environment of an on-unit is provided by the the block containing the ON statement establishing the on-unit.

A label constant designates a point within the text of a block, B. During execution, there may be several activations of B; it is essential to know the particular activation of B which is referred to by a label reference. A reference to a label constant L, made in some activation of a block B1, is to L in the current environment of B1. When a label constant is assigned to a label variable, this environmental information is assigned as well. Subsequent GO TO statements naming the label variable will re-establish the environment assigned to the variable, and hence may

31 MARCH 1972

cause blocks to be terminated. When a label variable is assigned to another label variable, the environmental information is assigned as well.

A generation, or allocation, of a variable is created whenever storage is allocated for the variable. A generation consists of the storage for the generation together with the evaluated set of attributes for the generation. Associated with the generation is a pointer to the storage allocation; this serves as a unique identification of the generation. The evaluated set of attributes is established when the generation is allocated and enables the contents of the storage to be interpreted. In some cases, the attributes may have to be re-evaluated upon each reference.

In the case of static and automatic generations, the pointer to the generation can only be obtained by invoking the built-in function ADDR using the variable as the argument. For based variables, a locator variable is specified in the ALLOCATE statement used to create the based variable, and a value is assigned to it so that it can be used to access the generation that is created.

The storage for a generation contains the values of the various fields in the variable. The evaluated set of attributes of a generation comprises the structuring of the variable, the data types of its components, and the bounds of arrays and lengths of strings as evaluated at the time of allocation. Offset variables may be used to identify the position of a generation within a file. If the offset and file reference are supplied as arguments of the POINTER built-in function, the result is a pointer identifying the generation. Similarly, if the pointer and file reference are supplied as arguments of the OFFSET built-in function, the result is the offset of the generation from the beginning of the file.

ARGUMENT PASSING

When a procedure is invoked, a relationship is established between the arguments of the invoking statement and the parameters of the invoked entry point. A procedure may pass one of its parameters as an argument to another procedure (or even to itself in a recursive call).

The ENTRY attribute must be used to specify the attributes of all arguments of an external procedure. The correspondence of parameters in a parameter list with the arguments in an argument list is from left to right, with the

31 MARCH 1972

left-most parameter corresponding with the left-most argument. The number of arguments and parameters must be the same. In addition, the attributes of each argument in a procedure reference must match the attributes of the corresponding parameter at the invoked entry point. When an argument is a subscripted variable, the subscripts are evaluated before invocation. The specified element is then passed as the argument. Subsequent changes in the subscript or the locator identifying the generation of the argument during the execution of the invoked procedure have no effect upon the corresponding parameter.

Parameters

The PROCEDURE and ENTRY statements may specify a list of parameters. Parameter lists for different entries to a procedure need not be the same. A parameter may be a scalar, array, or major structure name that is unqualified and unsubscripted. A reference within the procedure to a parameter produces an undefined result if the entry point at which the procedure is invoked does not include that parameter in its parameter list. Parameters are explicitly declared by their appearance in a PROCEDURE or ENTRY statement. Additional attributes must be supplied in a DECLARE statement internal to the procedure.

Parameters cannot be declared with the storage class attributes STATIC, AUTOMATIC, or BASED, or with the BUILTIN or INITIAL attributes. However, parameters may be declared with the storage class attribute REGISTER. Scope attributes cannot be declared for parameters; a parameter has internal scope. Any bounds or lengths must be specified either by asterisks or decimal integer constants which may be signed. If a parameter is a structure, it must be a major structure.

Example:

```

SBPRIM: PROCEDURE(X, Y, Z);
        DECLARE (X,Y,A,B) FIXED,
                Z FLOAT;
        A = X - 1;
        B = Y + 1;
        GO TO COMMON;
SBSEC:  ENTRY(X, Z);
        A = X - 2;
        B = X - 3;
COMMON:  Z = A**2 + A*B + B**2;
        END SBPRIM;

```

31 MARCH 1972

In the above example, the procedure SBPRIM may be entered at its primary entry point SBPRIM, where the parameter list is (X, Y, Z), or at its secondary entry point SBSEC, where the parameter list is (X, Z).

Correspondence of Arguments and Parameters

The number and data types of the arguments in a procedure reference must be the same as the number and data types of the parameters in the corresponding parameter list (where the parameter list is given in the PROCEDURE or ENTRY statement for internal procedures and in the ENTRY declaration for external procedures). This is true even if a dummy argument is constructed. The only exception to this rule is that the REGISTER attribute may be specified for an argument without being specified for the corresponding parameter, or it may be specified for a parameter without being specified for the corresponding argument. In the following example, dummy arguments will be constructed for the last two arguments because the corresponding parameters have the REGISTER attribute. (See "Use of Dummy Arguments" for implications.)

```

P1: PROCEDURE;
    DECLARE (A, B) FIXED REGISTER,
           (C, D) FIXED AUTOMATIC,
           P2   ENTRY (FIXED,
                     FIXED,
                     FIXED REGISTER,
                     FIXED REGISTER);
    ...
    CALL P2(C, A, B, D);
    ...
END P1;

```

If a parameter of an invoked entry is a scalar, the argument must be a scalar expression. The data attributes of the argument or dummy argument must agree with the corresponding attributes of the parameter. No data type conversion will be performed. However, arithmetic conversions may be performed in the invoking procedure if the scale and precision of an expression do not match the attributes declared for the referenced entry. If the bounds or lengths of parameters are explicitly declared, then they must match those of the corresponding arguments; however, if they are declared with asterisks (see "Dimension" and "BIT and CHARACTER" in Chapter 5), then they will automatically match. If the argument has the VARYING attribute, then the parameter must also be declared with this attribute.

31 MARCH 1972

If a parameter of an invoked entry is an array, the argument in general must be an array expression with identical bounds and dimensionality. If constants are used to specify the bounds of the parameter in the invoked procedure, the values of the bounds of the array argument must agree with the values of these constants.

If a parameter is a structure, the argument must be a structure or substructure. The data attributes of the elements of the argument structure must match those of the associated parameter as specified in the invoked procedure. The relative structuring of the argument and the parameter must be the same, although the level numbers need not be identical. Contained strings and arrays with lengths and bounds specified by constants must agree. The REFER attribute must not be used in a parameter declaration.

If a parameter is a scalar label variable, the argument must be a scalar label expression. If a parameter is an array label variable, the argument must be an array label expression. A dummy argument is always constructed when the argument is a label constant. This dummy argument will also contain identification of the current invocation of the block containing the label. Any reference to the parameter is a reference to the statement label in that environment.

If a parameter is an entry parameter, the corresponding argument must be an unparenthesized entry expression. The names of built-in functions or procedures may not be passed as entry constants.

Use of Dummy Arguments

A constructed dummy argument containing the argument value is passed to a procedure if the argument is one of the following:

- an expression involving operators
- an expression in parentheses
- a label constant
- an entry constant
- a function reference
- a scalar which requires arithmetic conversion

A dummy argument is also constructed if the corresponding parameter has the REGISTER attribute.

In all other cases the argument as it appears is passed. The parameter becomes identical with the passed argument, so that changes to the parameter are also changes to the passed

31 MARCH 1972

argument. However, if a dummy is created, changes to the parameter are not reflected back in the original argument.

Note that no dummy argument is created for an arithmetic or string constant. If an attempt is made to modify such an argument, an execution-time error will occur.

Entry References as Arguments

When an entry reference is specified as an argument to a procedure, one of the following applies:

1. If the name of the entry referred to in the argument is M, then, if the reference specifies an argument list of its own, it is recognized as a function reference; M is invoked and the value returned by M effectively replaces M and its argument list in the containing argument list. If the attributes of the returned value do not match the declared attributes of the argument, the program is in error.
2. If the entry reference appears without an argument list, but within an operational expression or within parentheses, then it is taken to be a function reference with no arguments.
3. If the entry reference argument appears without an argument list and not within an operational expression or parentheses, the entry reference itself is passed to the function or subroutine being invoked. In such cases, the entry reference is not taken to be a function reference, even if it is the name of a function that does not require arguments. In this circumstance, the entry reference must not appear in parentheses, or it will be treated as case 2 above.

Example:

```

A:
  PROCEDURE;
    DECLARE B ENTRY RETURNS(FLOAT),
           C ENTRY(FLOAT);
    ...
    CALL C((B));
    ...
  END A;
```

31 MARCH 1972

In the CALL statement in this example, the entry B is invoked and the value returned by B is passed to C as an argument.

Use of the ENTRY Attribute

If an ENTRY attribute without a parameter attribute list is specified for an identifier, it indicates that the named entry does not require any arguments. In this case, it is an error to supply arguments in a reference to the entry. If an ENTRY attribute specification with a parameter attribute list is supplied for an identifier, each reference to the identifier that implies an invocation of the associated procedure must supply an argument list whose elements are identical in data type to those specified for the corresponding parameter. If there is disagreement, a compile time error message will be given. The asterisk notation may be used in the ENTRY attribute to specify that the bounds of arrays or strings are to be taken from the argument attributes.

While no data type conversions will be performed as a result of a procedure CALL or function reference, arithmetic conversions will be performed when required. If the scale or precision of an argument expression does not match the attributes for the referenced entry, an arithmetic conversion may take place. No conversions will be performed for data aggregates.

31 MARCH 1972

CHAPTER 3: DATA ELEMENTSINTRODUCTION

Information that is operated on during the execution of an Apple object program is called data. Each data item has a definite type and representation. The discussion on data elements presents:

1. the types of data available in Apple,
2. the various organizations of data, and
3. the methods by which data can be referenced.

DATA TYPES

The types of data allowed by Apple can be categorized as problem data and program-control data. Each category comprises both constants and variables.

A constant is a data item that denotes a value that cannot change during the execution of a program. The attributes of a constant are implied by the representation of the constant itself. A signed constant is an arithmetic constant preceded by one of the prefix operators + or -. Wherever the word "constant" appears alone, and refers to an arithmetic constant, it is to be assumed to refer to an unsigned constant.

A variable is a name given to a single data element (called a scalar variable) or a collection of data elements (called an array variable or a structure variable). The attributes of a variable are:

1. explicitly declared,
2. declared by the context in which the variable appears, or
3. assumed by default.

PROBLEM DATA

Problem data is any data that can be classified as type arithmetic or type string.

31 MARCH 1972

Arithmetic Data

An arithmetic data item is defined to have a numeric value with attributes of scale and precision. Arithmetic data items are real values and are represented internally in a binary format. Arithmetic constants may be expressed in decimal or hexadecimal but are internally represented as binary values. The attributes of an arithmetic data item are given by specifying scale (fixed or float) and precision (expressing the minimum number of binary or decimal digits to be maintained). These attributes determine the form of the internal representation of the data.

Scale

Arithmetic data may be specified as having either fixed-point or floating-point scale. Fixed-point data items are restricted to integers and have no associated scale factor. Floating-point data items are rational numbers consisting of a fractional part and an exponent part. The exponent part specifies the decimal or binary point location.

Precision

The precision of arithmetic data items is either short (23 bits of precision) or long (47 bits of precision). The precision of arithmetic variables is specified through the use of the precision (BINARY and DECIMAL) attributes in the DECLARE statement. The general rules for the declared precision versus the internal precision are as follows:

Declared Precision	Resulting Precision
BINARY (1 to 23)	short
BINARY (24 to 47)	long
DECIMAL (1 to 6)	short
DECIMAL (7 to 14)	long

31 MARCH 1972

Example:

	Resulting Precision
DECLARE A FIXED BINARY(15),	short
B FIXED DECIMAL (5),	short
C FIXED BINARY(31),	long
D FLOAT DECIMAL (7),	long

Note that the number of binary or decimal digits must be greater than zero. If the number of digits specified exceeds the limit of precision stated above, the maximum is assumed and a diagnostic message is produced.

The range of values that can be represented by arithmetic data depends on the scale and precision of the data items:

<u>Scale and Precision</u>	<u>Range of Values</u>
FIXED short	± 8,388,607
FIXED long	± 140,737,488,355,327
FLOAT short	± 10 ^{±33}
FLOAT long	± 10 ^{±8630}

31 MARCH 1972

Arithmetic Constants

The general form of arithmetic constants is as follows:

```

arithmetic-constant ::= decimal-number |
                        hexadecimal-number

decimal-number ::= [sign] integer. [integer][exponent]|
                 [sign][integer]. integer [exponent]|
                 [sign] integer[exponent]

integer ::= decimal-digit...
exponent ::= E [sign] integer
hexadecimal-number ::= "hexadecimal-digit ... "
```

Examples:

```

123
+45
"ABC"
123.4E+02
.31
-42E+3
```

The scale and precision of hexadecimal constants are implied by the number of hexadecimal digits represented:

<u>No. of hex.digits</u>	<u>Scale and Precision</u>
1 to 6	FIXED short
7 to 8	FLOAT short
9 to 12	FIXED long
13 to 16	FLOAT long

String Data

A string is a contiguous sequence of characters or binary digits that can be treated as a single data item. String data can be classified as character-string or bit-string. All strings have an associated length attribute which is declared for string variables and implied for string constants. The maximum length allowed for string data in the Apple implementation is 65,535 bits or characters.

31 MARCH 1972

Character-String Data

Character-string data consists of a string of zero or more characters in the data character set. The string may be fixed or varying in length. The actual number of characters must be specified if it is of fixed length, and the maximum length must be specified if it is of varying length.

Note: Until a varying-length character-string is assigned a value, its length is undefined.

A comment will not be recognized within a character string, but will be considered to be part of the character string data including the comment delimiters (/ * and * /).

Character-String Constants

A simple character-string constant is zero or more characters in the data character set enclosed in single quotation marks. If it is desired to represent a quotation mark, it must be represented as two immediately adjacent single quotation marks, although it is only counted as a single character.

Examples:

```
'$123.45'  
'JOHN JONES'  
'IT''S'  
''
```

The last example, which is two single quotation marks with no intervening blank, specifies the null character string. In the Apple implementation, character-string data is maintained internally in ASCII character format, in which each character occupies one byte of storage. (See Appendix 4 for the Apple character set.) A simple character-string constant may optionally be preceded by an unsigned decimal integer constant in parentheses to specify repetition. If the constant specifying repetition is zero, the result is the null character string.

Example:

(3)'TOM_' is exactly equivalent to 'TOM_TOM_TOM_'

31 MARCH 1972

Bit-String Data

Bit-string data consists of a string of zero or more binary digits (0 and 1). The bit-string must be fixed in length.

Bit-String Constants

A bit-string constant contains zero or more binary digits enclosed in single quotation marks, followed by the letter B. A bit-string constant may also be written as a string composed of hexadecimal digits enclosed in single quotation marks and followed by the letter H. In this latter case, each digit represents 4 bits. The repetition factor as described for character-string constants may also precede bit-string constants.

Examples:

```
'11101'B
    'B
'015BD7'H
is exactly equivalent to
'000000010101101111010111'B
```

PROGRAM-CONTROL DATA

Program-control data is any data that can be classified as label, locator, interrupt, file, or entry.

Label Data

Statement label data is used only in connection with statement labels. Statement label data may be constants or variables, and the variables may be elements of structures or arrays.

Statement-Label Constants

A statement label constant is an unsubscripted identifier that precedes the statement with a colon separating the statement and the statement label. It permits references to be made to statements.

31 MARCH 1972

Example:

```

ROUTINE1:  ...
           IF X > 5 THEN
             GO TO DONE;
           ...
           GO TO ROUTINE1;
DONE:      ...
           RETURN;

```

ROUTINE1 and DONE are statement-label constants.

Statement-Label Variables

A statement-label variable is a variable that has as values statement-label constants. These variables can be grouped into arrays, or they may be elements of structures.

Example:

```

DECLARE X LABEL VARIABLE;
X = POSROUTINE;
...
POSRoutine: ...
...
X = NEGROUTINE;
GO TO X;
...
NEGROUTINE: ...
...

```

The label variable X may have the value of either POSROUTINE or NEGROUTINE. In the above example, GO TO X; transfers control to NEGROUTINE.

Locator Data

A locator value identifies a specific generation of a based variable. Since several generations of a based variable can exist simultaneously, a reference to a based variable must include, either explicitly or implicitly, a locator variable whose value defines the actual generation being referenced. Locator data consists of pointer variables, offset variables and descriptor variables.

A pointer variable identifies a generation of a based variable within a program and is only valid while the program is active.

31 MARCH 1972

An offset variable identifies a generation of a based variable relative to the origin of a file and thus preserves its validity independent of the program.

Neither pointer nor offset variables contain any information concerning the attributes of the based variable being referenced other than location. Descriptor variables, in addition to containing a pointer value, also contain the length of the based variable identified. If the based variable is a character or bit string, then the length is in terms of characters or bits respectively. If the based variable is a vector of arithmetic elements, then the length is the number of elements in the vector.

Locator variables may have values set by the ALLOCATE, FIND, and LET statements or by assignment from other locator variables or from the ADDR, NULL, POINTER, APLEVAR, OFFSET, and DESCR built-in functions. In addition, descriptor variables may be used in arithmetic expressions. Pointer and offset variables may not be used as operands in any expression other than = and -= comparison.

Note: Descriptor variables have been added to the Apple language to support systems programming and provide a high-level language facility for utilizing the data-streaming capabilities of the STAR computer.

Locator Qualification

Locator qualification is used to associate one or more descriptor, pointer or offset values so as to identify a particular generation of data. If a based variable is referred to without a locator qualifier, the reference is the same as a reference qualified by the locator variable declared with the based variable in the BASED attribute specification.

General format:

locator qualifier ::= scalar-locator-expression->

[based-locator-variable->]... based-variable

where "scalar-locator-expression" is an descriptor-variable, a pointer-variable, an offset-variable, or a function reference that returns a descriptor, pointer, or offset value.

31 MARCH 1972

General rules:

1. Locator qualification is used to identify the generation of a based variable to which the associated reference applies.
2. If an offset expression or an offset variable is used as a locator qualifier, its value is implicitly converted to a pointer value.
3. If more than one qualifier is used, they are evaluated from left to right.

Examples:

```
A = P -> B;
A = P -> Q -> B;
A = ADDR(X) -> B;
```

The first example causes assignment to A of the value of B in the generation pointed to by P. The second example specifies that the value of P is to be used to locate the generation of Q which locates the specific generation of B to be assigned to A. In the third example, the generation of B is derived from the location of the variable X.

Interrupt Data

An interrupt is an action which can discontinue normal execution of a program. There are two types of interrupts, conditions and events. A condition is raised by the occurrence of an error as a result of an instruction execution and may be thought of as internal to a program, while an event is an external action that can occur on a peripheral device. The execution of the SIGNAL statement will also cause an interrupt. When an interrupt occurs, the associated condition is raised or the event is completed. See CONDITIONS and EVENTS in Chapter 7.

File Variable

A file is a collection of data that occupies memory and, through the use of a MCTS file management function, may be stored on a peripheral storage device. After a file has been opened it may be referenced through a file-variable. A file-variable may be used in the GET/PUT or ALLOCATE/FREE statements in order to reference a particular file. See Chapter 6 for a description of file handling.

31 MARCH 1972

Entry Data

Entry data has values that permit references to be made to entry points of a program. Entry data may be constants or variables. An entry constant is an identifier that appears in a program as an entry name. It permits references to be made at a fixed entry point of a procedure. An entry variable has entry constants as values. See "The ENTRY Attribute" in Chapter 5.

31 MARCH 1972

ORGANIZATION

Data may be organized as scalar items (i.e., single data items) or aggregates of data items (i.e., arrays and structures).

Scalar Items

A scalar item may be either a constant or the value of a scalar variable. Constants and scalar variables are called scalar data items.

Scalar Variables

A scalar variable is a single data item. Unlike a constant, however, a variable may take on more than one value during the execution of a program. The set of values that a variable may take on is the range of the variable. The range of a variable is always restricted to one data type and, if the type is arithmetic, to one scale and precision -- see "Arithmetic Data" in this chapter.

Reference is made to a scalar variable by a name, which may be a simple name, a subscripted name, a qualified name, or a subscripted qualified name (see "Naming" in this chapter).

Data Aggregates

In Apple, all classes of variable data items except ENTRY and ENTITY may be grouped into arrays or structures. Rules for this grouping are given below. For the method of referring to an array or structure or a particular item of an array or structure, see "Naming" in this chapter.

Arrays

An array is an multi-dimensional, ordered collection of elements, all of which have identical data attributes. (If arithmetic, all of the elements of the array must have the same scale and precision. If character-string or bit-string, all of the elements must have the same fixed length or the same maximum length.) The number of dimensions of an array, and the upper and lower bounds of each dimension, are specified by the use of the dimension attribute. (See "The Dimension Attribute" in Chapter 5.) The elements of an array may be structures (see "Arrays of Structures" in this

31 MARCH 1972

chapter).

Structures

A structure is a hierarchical collection of scalar variables, arrays and structures. These need not be of the same data type nor have the same attributes.

The outermost structure is a major structure, and all contained structures are minor structures.

A structure is specified by declaring the major structure name and following it with the names of all contained minor structures and base elements. Each name is preceded by a level number, which is an unsigned non-zero decimal integer constant. A major structure is always at level one and all minor structures and base elements contained in a structure (at level n) have a level number that is numerically greater than n, but they need not necessarily be at level n+1, nor need they all have the same level number.

A minor structure at level n contains all following items declared with level numbers greater than n up to but not including the next item with a level number less than or equal to n. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a DECLARE statement.

31 MARCH 1972

Example:

```

DECLARE  1 PAYROLL,
          2 NAME CHAR(8) ,
          2 HOURS,
            4 REGULAR FIXED,
            3 OVERTIME FIXED,
          2 JOBS,
            3 NUMBER(2) FIXED,
            3 DESCRIPTION(2) FIXED,
          2 RATE FIXED;

```

In the above example PAYROLL is defined as the major structure containing the scalar variables NAME and RATE and the structures HOURS and JOBS. The structure HOURS contains the scalar variables REGULAR and OVERTIME. Note that REGULAR and OVERTIME are at the same level although their level numbers differ. The structure JOBS contains NUMBER and DESCRIPTION which are both one-dimensional arrays with two scalar variables.

Arrays of Structures

An array of structures is specified by giving the dimension attribute to a structure, thus forming replications of that structure. Each element of the array is one instance of the declared structure. The elements within an array of structures must be referred to by subscripted names (see NAMING in this Chapter).

Example:

```

DECLARE  1 CARDIN(3) ,
          2 NAME CHAR(8) ,
          2 WAGES,
            3 NORMAL FIXED,
            3 OVERTIME FIXED;

```

The name CARDIN represents an array structures of with bounds 1:3. Note that each of the three structures formed by CARDIN(3) has an element called NAME, WAGES.NORMAL, and WAGES.OVERTIME. Each of these elements must have a subscript with the name to indicate which structure is desired.

31 MARCH 1972

Attributes of Structures

Structures and arrays of structures are not given data attributes. These can be given only to scalar variables or arrays forming the elements of major or minor structures.

Major structure names may be declared with scope and storage attributes. Items contained in structures may not be declared with these attributes. When the same major structure name is declared with the EXTERNAL attribute in more than one block, the attributes of the structure members must be the same in each case, although the names of the structure members need not be the same. A reference to a member in one such block is effectively a reference to that member in all blocks in which the external name is known, regardless of the names of the members.

Since only the major structure may be given a storage-class attribute, all items in the same structure are of the same storage class. The storage class of the major structure applies to all elements of the structure. If a structure has the BASED attribute, only the major structure, not its elements, may be allocated and freed.

NAMING

This section describes the rules for referring to a particular data item, groups of items, arrays, and structures. The permitted types of data names are: simple, qualified, subscripted, and subscripted qualified.

Simple Names

A simple name is an identifier (see "Identifiers" in Chapter 1) that refers to a scalar, an array, or a structure.

Subscripted Names

A subscripted name is used to refer to an element of an array. It is a simple name that has been declared to be the name of an array followed by a list of subscripts. The subscripts are separated by commas and are enclosed in parentheses. A subscript is an scalar arithmetic expression converted to an integer before its use. The number of subscripts must be equal to the number of dimensions of the

31 MARCH 1972

array, and the value of a specified subscript must fall within the bounds declared for that dimension of the array.

General formats:

```
subscripted-name ::=
    identifier (subscript[, subscript]...)

subscript ::= scalar-expression
```

Examples:

```
A(3)
FIELD(B, C)
PRODUCT(SCOPE*UNIT*VALUE, PERIOD)
ALPHA(1, 2, 3, 4)
```

Qualified Names and Ambiguous References

A simple name usually refers uniquely to a scalar variable, an array, or a structure. However, it is possible for a name to refer to more than one variable, array, or structure if the identically named items are themselves parts of different structures. In order to avoid any ambiguity in referring to these similarly named items, it is necessary to create a unique name; this is done by forming a qualified name. This means that the name common to more than one item is preceded by the name of the structure in which it is contained. This, in turn, can be preceded by the name of its containing structure, and so on, until the qualified name refers uniquely to the required item.

Thus, the qualified name is a sequence of names, separated by periods, specified left to right in order of increasing level numbers. The sequence of names need not include all of the containing structures, but it must include sufficient names to resolve any ambiguity. Any of the names may be subscripted.

If the sequence of names includes the names of all the structures containing the member with the rightmost name, then that name is said to be completely qualified.

If the sequence of names includes only some of the names of the structures containing the member with the rightmost name, then that name is said to be partly qualified.

31 MARCH 1972

A completely or partly qualified name must have the same hierarchy of the structure names as the structure to which it is to reference. The qualified name, once composed, is itself a name. Subsequently, in this publication, when the terms scalar variable name, array name, or structure name are used they should also be taken to include qualified names.

General format:

```
qualified-name ::= identifier[.identifier]...
```

There are several rules that should be followed when using qualified names. (In the following examples the attributes have been eliminated for clarity.) These are as follows:

1. The qualified name will resolve to the innermost block containing the declaration which has the same hierarchy of the identifiers as the qualified name. That is, if the name cannot be resolved in the block of its usage, then the next outer block will be checked, etc. A diagnostic message results if the qualified name cannot be resolved.

Example:

```

DECLARE 1 A,
        2 C,
        2 D,
        3 E;
BEGIN;
  DECLARE 1 A,
          2 B,
          3 C,
          3 E;

```

A.C refers to C in the inner block.

D.E refers to E in the outer block.

A.B.D is in error.

2. If there is more than one structure declaration in the same block which contains the same qualified name then only one of these declarations may contain the completely qualified name.

31 MARCH 1972

Example:

```

DECLARE    1 A,
           2 B,
           3 C;
DECLARE    1 A,
           2 D,
           3 B;

```

A.B refers to the first declaration

A.D.B refers to the second declaration

3. A reference to a structure member by means of an unqualified name is ambiguous and therefore in error if any other structure member name internal to the same block has the same identifier.

The case where more than one declaration contains the same qualified name is illustrated in the following:

Example:

```

DECLARE    1 A,
           2 B,
           3 C;
DECLARE    1 A,
           2 D,
           3 C;

```

A.C is ambiguous because neither C is completely qualified by this reference.

The case where a single declaration contains multiple occurrences of the same qualified name is illustrated in the following:

31 MARCH 1972

Example:

```

DECLARE    1 Y,
           2 X,
           3 Z,
           3 A,
           2 Y,
           3 Z,
           3 A;

```

Y.Z is ambiguous and in error.
 Y.Y.Z refers to the second Z.
 Y.X.Z refers to the first Z.

4. If a level-1 name and a structure member name internal to the same block have the same identifier, then the unqualified use of that identifier is taken to refer to the level-1 name. Reference to the structure member can, in this case, be achieved only by means of a suitably qualified name.

Example:

```

DECLARE    1 A,
           2 A,
           3 A;

```

A refers to the first A.
 A.A refers to the second A.
 A.A.A refers to the third A.

Subscripted Qualified Names

The elements of an array contained in a structure and requiring name qualification for identification are referred to by subscripted qualified names. A subscripted qualified name is a sequence of names and subscripted names separated by periods. The order of names is as given for any qualified name. The subscript list following each name refers to the dimensions associated with the name if the name is declared to be the name of an array in the structure description.

As long as the order of the subscripts remains unchanged, subscripts may be moved to the right or left (called migration of subscripts) and attached to names at a lower or higher level. The number of subscripts must match the number of dimensions of the array.

31 MARCH 1972

General format:

subscripted-qualified-name ::=

```

    identifier[ (subscript[ , subscript]...) ]

```

```

    [.identifier[ (subscript[ , subscript]...) ]]....

```

If any subscripts are given in a reference to a qualified name, all those subscripts which apply to dimensions of containing structures must be given.

Example:

A is an array of structures with the following description:

```

DECLARE    1 A (10,12),
           2 B (5),
           3 C (7),
           3 D;

```

The following subscripted qualified names illustrate the migration of subscripts referring to the same element, which is the seventh element of C contained in the fifth element of B contained in the tenth row and twelfth column of A:

```

(1) A (10,12) . B (5) . C (7)
(2) A (10) . B (12,5) . C (7)
(3) A (10) . B (12) . C (5,7)
(4) A . B (10,12,5) . C (7)
(5) A . B (10,12) . C (5,7)
(6) A . B (10) . C (12,5,7)
(7) A . B . C (10,12,5,7)
(8) A (10,12) . B . C (5,7)
(9) A (10) . B . C (12,5,7)
(10) A (10,12,5,7) . B . C

```

If structure B, but not structure A, is necessary for unique identification of this use of C, any of forms (4), (5), (6), or (7) may be used without including the A.

If structure A, but not B, is necessary for identification of C, forms (7), (8), (9), or (10) may be used without including the B.

31 MARCH 1972

CHAPTER 4: DATA MANIPULATIONINTRODUCTION

This chapter describes the two main areas of data manipulation:

1. expression evaluation
2. data conversion

The first section describes the logical classes of expressions and the operations available in each class. The second section specifies the data conversion rules to be used for data type conversion and arithmetic conversion.

EXPRESSIONS

An expression is a representation of a value or an algorithm used for computing a value. Expressions are generally classified according to the type and form of the data values they represent. If an expression represents a single scalar value, it is called a scalar expression. An array expression represents an array of values.

Problem data values are represented by arithmetic expressions and string expressions. Arithmetic expressions whose value is fixed point are known as integer expressions. Expressions representing program-control data values are similarly defined. Thus, a pointer expression is an expression that represents a pointer value, whereas a locator expression may represent either a pointer, descriptor, or offset value.

In the syntactic descriptions used in this manual, the unqualified term "expression" refers to an expression of any type. Where the kind of expression is limited, the type of restriction is explicitly noted; for example, "scalar expression" indicates that only an expression that represents a scalar value is permitted in the particular context.

31 MARCH 1972

Expressions may also be classified by the operators that they contain. An expression containing operators (either prefix or infix operators or both) is referred to as an operational expression. The class of an operational expression is determined by the class of operators it contains. The four classes of operational expressions are:

CLASS of operational expressions	OPERATORS	DATA TYPES permitted as operands
ARITHMETIC	** , prefix + and - , * and / , infix + and -	FIXED FLOAT
DESCRIPTOR ARITHMETIC	infix + and -	DESCRIPTOR
RELATIONAL	< , <= , = , >= , > , >	FIXED , FLOAT CHARACTER DESCRIPTOR
	only = and <=	BIT , LABEL POINTER , OFFSET FILE , ENTRY
BIT STRING	~ & !	BIT Relational- expressions
STRING	!!	CHARACTER , BIT

An expression may be:

1. a constant
2. a reference to a variable
3. a function reference
4. an expression enclosed in parentheses
5. an expression preceded by a prefix operator
6. two expressions connected by an infix operator

There is no limit to the number of operators and level of parentheses that may be combined in a single expression. Generally, all of the operands contained in a single

31 MARCH 1972

expression must be of the same type (FIXED and FLOAT are considered to be the same type for this purpose) and all of the operators within the expression must be of the same class. No implied data type conversion can occur during the evaluation of an expression. If the operands are not of matching data type, the necessary conversion may be explicitly specified by using the built-in functions for conversion, for example, FIXED, FLOAT, CHAR, etc. These are defined in Appendix 1.

Arithmetic operations

An elementary arithmetic operation has the following general format:

$$\left. \begin{array}{l} \phantom{\text{operand}} \{ + \mid - \} \text{ operand} \\ \text{operand} \{ + \mid - \mid * \mid / \mid ** \} \text{ operand} \end{array} \right\}$$

The general format specifies the prefix operations of plus and minus and the infix operations of addition, subtraction, multiplication, division, and exponentiation.

Any result of the prefix operations has the same scale and precision as the operand. If both operands of an infix operation (+, -, or *) are FIXED, the scale of the result is also fixed-point; otherwise, the operation is performed in floating-point and the result is FLOAT. The precision of all infix operations is the greater of the precisions of the two operands. Any necessary conversion of FIXED to FLOAT or short to long precision is performed before the infix operation is carried out. The details of arithmetic data conversion are described later in this chapter.

An exception to the scale conversion rule occurs in the case of exponentiation. If the scale of the first operand is float and the exponent operand is a fixed expression, no conversion is necessary. The result will be floating-point.

31 MARCH 1972

An arithmetic expression of any complexity is composed of a combination of elementary arithmetic operations defined above. The evaluation of compound arithmetic expressions is performed in the following order of decreasing operator precedence (unless the order is modified by parentheses):

1. ** and prefix + operators are performed right to left
2. * and / operations are performed from left to right,
3. Infix + and - operations are performed left to right.

Thus,

$$A + B ** - C / D - E$$

is performed as

$$(A + ((B ** (-C)) / D)) - E$$

The infix operators, + and *, are commutative, but not necessarily associative, as low-order rounding errors will depend on the order of evaluation of an expression. Thus, $A + B + C$ is not necessarily equal to $A + (B + C)$.

Prefix operators can precede and be associated with either of the operands of an infix operation. For example, in the expression $A * - B$, the minus sign preceding the variable B indicates that the value of A is to be multiplied by the negative value of B.

More than one prefix operator can precede and be associated with a single variable. More than one positive prefix operator will have no cumulative effect, but two consecutive negative prefix operators will have the same result as a single positive prefix operator.

Descriptor Arithmetic

Descriptor expressions have the following form:

$$\text{descriptor-variable} \left[\begin{array}{c} \{ + \\ - \\ * \} \end{array} \text{fixed-point-expression} \right]$$

The result of the expression will be a descriptor whose length value is taken from the descriptor variable and whose pointer value is the fixed-point result of the specified

31 MARCH 1972

operation between the descriptor used as a fixed value and the fixed-point-expression.

Relational Operations

Elementary relational operations have the general form:

$$\text{operand} \left(\begin{array}{l} < \\ \neg < \\ < = \\ = \\ \neg = \\ > = \\ > \\ \neg > \end{array} \right) \text{operand}$$

There are five kinds of relational comparison:

1. Arithmetic involves the comparison of signed numeric values, possibly obtained by the evaluation of expressions. If the operands differ in scale or precision, they are converted before the comparison is made (see "Arithmetic Conversion" later in this chapter).
2. Descriptor comparisons are made by comparing the pointer values as fixed-point data. The length values are ignored. Thus, two descriptors that identify the same based variable but have different length values will compare equal. Descriptors can be compared with fixed-point, descriptor, or pointer expressions.
3. Character involves left-to-right, character-by-character comparisons of characters according to the collating sequence defined in Appendix 4. If the operands are of different lengths, the shorter string is extended to the right with blanks. Two null character strings compare equal.
4. Bit involves the left-to-right comparison of binary digits. If the strings are of different lengths, the shorter string is extended on the right with zeros. Only equal and not-equal comparisons can be made between bit-string operands. Two null bit strings compare equal.

31 MARCH 1972

5. Program-control data involves the comparison of two data values from one of the following data types:
- a. statement label
 - b. pointer
 - c. offset
 - d. file
 - e. entry

Only the operators = and != may be used in this context and both operands must be of the same type as defined above. The comparison of two offset values is performed independently of their associated files. For two statement labels to compare equal, they must refer to the same statement within the same environment (see "The Environment of a Block" in Chapter 2).

The result of a relational operation is a true or false value, commonly used in the IF statement to select a conditional branch path. If necessary, the result of a relational comparison will be converted to a bit-string of length one; the value is '1'B if the relationship is true, or '0'B if the relationship is false.

Compound relational expressions are formed by combining elementary relational expressions as operands with the bit-string operators ~, & and !. See the "Example of Expression Evaluation" later in this chapter.

Bit-string Operations

Bit-string operations have the following general forms:

$$\left\{ \begin{array}{l} \sim \text{operand} \\ \text{operand} \& \text{operand} \\ \text{operand} ! \text{operand} \end{array} \right\}$$

The "not" operator can be used as a prefix operator only. The "and" and the "or" operators can be used as infix operators only. (These operators have the same function as in boolean algebra).

Operands of a bit-string operation must be bit strings or relational expressions that have been evaluated before the operation is performed. If the operands of an infix operation are of unequal length, the shorter is extended on the right with zeros to the length of the longer. The result of a bit-string operation is a bit string equal in

31 MARCH 1972

length to the length of the operands. The operations are performed from left to right on a bit-by-bit basis starting with the left-most bit of each string. As a result of the operations, each bit position has the value defined in the following table:

A	B	-A	-B	A & B	A ! B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

More than one bit-string operation can be combined in a single expression that yields a bit-string value. There are no varying-length bit strings.

String Operations

String operations have the following general form:

operand !! operand

The concatenation operator can be used as an infix operator between two character string operands or between two bit-string operands. It signifies that the operands are to be joined in such a way that the last character or bit of the first operand will immediately precede the first character or bit of the second operand. The length of the result is always the sum of the lengths of the operands. If either of the operands of the concatenation operator is a character string with the VARYING attribute, the result will also be a varying string. When varying strings are concatenated, the intermediate string created has a length equal to the sum of the maximum lengths. If the maximum lengths are known at compile time and their sum exceeds 65535, then a truncated intermediate string of length 65535 will be created and a compile-time diagnostic message produced. If the maximum length of either operand is not known at compile time and their sum exceeds 65535, a truncated intermediate string of length 65535 will be created but there will be no diagnostic message.

31 MARCH 1972

EVALUATION OF EXPRESSIONS

An operational expression may contain arbitrarily many different combinations of operands and operators provided that no implicit data type conversions are required of any operands or intermediate results. Generally, all of the operands will be of the same data type and the operators will belong to the same class of operators (this classification is shown in the table in the section "Expressions", earlier in this chapter). There are two exceptions to this rule:

1. Bit-string concatenation may be used with the logical bit-string operations, e.g.

```
BITA !! BITB & BITC
```

2. Compound relational expressions may contain relations that compare different data types, e.g.

```
IF (FIXED = 5) & (CHAR4 = 'THIS') THEN...
```

Each operation within the expression is evaluated according to the rules for that kind of operation. However, the order in which the sub-expressions are evaluated depends upon the priority of the operators specified in the expression.

Priority of Operators

The following table lists the seven levels of priority of operators in descending order. Each line lists the operators of the same priority level.

Priority level	Operators	Order of evaluation within this level
Highest 7	~, **, prefix+, prefix-	Right-to-left
6	*, /	Left-to-right
5	infix+, infix-	
4	!!	
3	<, <~, <=, <=~, =, >=, >~, >	
2	&	
Lowest 1	!	

31 MARCH 1972

Operations within an expression are performed in the order of decreasing priority. For example, in the expression $A+B**X$, the exponentiation is performed before addition.

Use of Parentheses

The order of evaluation of the sub-expressions of an expression can be changed by the use of parentheses. If a sub-expression is enclosed in parentheses, it indicates that the sub-expression is to be treated as a single value in relation to its adjoining operators. For example, in the expression:

$$(A + B**3) / (C * (D - E))$$

A will be added to $B**3$, E is subtracted from D before multiplying by C, and then the first of these results will be divided by the second result. Thus, parentheses modify the normal rules of priority.

The Apple implementation may evaluate subscripts, function references, and locator qualifiers in any order subject to the constraint that an operand will be fully evaluated before its value is used in an operation.

Example of Expression Evaluation

The following example of a compound relational expression illustrates how many operators and data types may appear in a single expression:

```
L1: IF A ** -B > C + D / COS(E)
      ! ~(BIT4 !! BITX) = (BITX & "F")
      ! LABEL_VARBL ^= L1
      & CHARX = CHAR3 !! SUBSTR(CHAR5, I+2, L/3)
      THEN DO; ...
```

The expression contains four elementary relational expressions (shown on separate lines for clarity) whose operands are themselves expressions. The first relational operation compares two arithmetic expressions, the second one compares two bit-strings, the next one tests a label variable and a label constant for inequality, and the final one compares a

31 MARCH 1972

character string with a character string expression. The function references are evaluated before their values can be used as an operand. The elementary relations within a compound relational expression may be evaluated in any order. The following list of steps describes one possible order of evaluation for this expression:

1. In the expression $A ** -B$, the minus sign is a prefix operator and thus has the same precedence as $**$, therefore the operations are evaluated in right-to-left order. The result follows the normal algebraic convention of raising A to the power $-B$.
2. The second operand of the first relation,

$$C + D / \text{COS}(E)$$
 is evaluated by computing the value of the cosine of E , dividing D by this value and then adding the result to C .
3. The result of step 1 is then compared to the result of step 2. If the first value is arithmetically greater than the second one, the relation is true and control will transfer to the DO statement of the THEN clause since the entire compound relational expression is true. Otherwise, if the value of the first relational operation is false, evaluation continues with the next step.
4. The value of $\sim (\text{BIT4} !! \text{BITX})$ is formed by concatenating the bit-strings BIT4 and BITX and then complementing the result.
5. The expression $(\text{BITX} \& \text{"F"})$ is evaluated by performing the $\&$ operation between the bit string BITX and the constant quantity "F" , with the shorter string extended on the right with zeros. Note that the parentheses are needed here since the relational $=$ operator has a higher priority than the $\&$ operator.
6. The $=$ comparison of the two bit-string results is made after extending the shorter string with zeros. If the relation is true, control will be transferred to the THEN clause for the same reason as in step 3. If this relational operation gives the value false, evaluation continues.
7. The third relational operation compares the value

of the label variable LABEL_VARBL with the label constant L1. This comparison involves checks of whether the two label values refer to the same statement and whether the environment indicator in the label variable refers to the current environment. If both these comparisons are true then the \neq relation is false and control will branch around the THEN clause since the entire compound relation is then false. Otherwise, the two label values are unequal and the relation holds true and evaluation continues.

8. Expressions within an argument list are evaluated before the corresponding function reference can be made; therefore, in the function reference


```
SUBSTR(CHAR5, I+2, L/3)
```

 I+2 and L/3 are computed before performing the substring function.
9. The substring extracted from CHAR5 by the SUBSTR function is concatenated to the right-hand end of the character-string, CHAR3.
10. The resulting character-string is compared for equality with the character-string CHARX after the shorter string has been extended with blanks. If the relation is true, the THEN clause is executed. Otherwise, control is transferred around the do-group of the THEN clause.

As the example was written above, the THEN clause will be executed if any of the three conditions are satisfied:

1. the first relation is true.
2. the first relation is false but the second relation is true.
3. both the first and second relations are false but both the third and fourth relations are true.

31 MARCH 1972

ARRAY EXPRESSIONS

A single array variable or an expression that includes at least one array operand is called an array expression. Array expressions may also include operators (both prefix and infix), scalar variables and constants.

Evaluation of an array expression yields an array result. All operations performed on arrays are performed on an element-by-element basis, in row-major order. Therefore, all arrays referred to in an array expression must be of identical bounds. Since the operations are performed on a strict element-by-element basis, array operations do not always produce the same result as the same operation in conventional matrix algebra.

Array expressions can be used only on the right-hand side of an assignment statement or as arguments. An array expression cannot appear in the relation of an IF statement. In this context, only an element expression can be valid since the IF statement tests a single true or false result.

Operations between Arrays and Scalars

The result of an infix operation between an array and a scalar element is an array with bounds identical to the original array, each element of which is the result of the operation being performed upon the corresponding element of the original array and the single element. For example:

If A is the array	$\begin{bmatrix} 5 & 10 & 3 \\ 12 & 11 & 8 \end{bmatrix}$
then $A*3$ is the array	$\begin{bmatrix} 15 & 30 & 9 \\ 36 & 33 & 24 \end{bmatrix}$

The element of an operation between an element and an array can be an element of the same array. In this case, the value used for the element throughout the operation is the value of the element before the start of the operation. For example, the expression $A*A(1, 3)$ would give the same result in the case of the above array A, since the original value of $A(1, 3)$ was 3.

31 MARCH 1972

Operations between Arrays

If two arrays are connected by an infix operator, the two arrays must have identical bounds. The result is an array with bounds identical to those of the original arrays; the operation is performed upon the corresponding elements of the two arrays.

Note that the arrays must have identical bounds. They must have the same number of dimensions; corresponding dimensions must have identical lower and upper bounds. For example, the bounds of an array declared X(10, 6) are not identical to the bounds of an array declared Y(2:11, 3:8), although the extents are the same for corresponding dimensions and the number of elements is the same.

Example of an array infix expression:

If A is the array	$\begin{bmatrix} 2 & 4 & 3 \\ 6 & 1 & 7 \\ 4 & 8 & 2 \end{bmatrix}$
and if B is the array	$\begin{bmatrix} 1 & 5 & 7 \\ 8 & 3 & 4 \\ 6 & 3 & 1 \end{bmatrix}$
then A*B is the array	$\begin{bmatrix} 2 & 20 & 21 \\ 48 & 3 & 28 \\ 24 & 24 & 2 \end{bmatrix}$

31 MARCH 1972

DATA CONVERSION

This section is concerned primarily with the concepts of data conversions, when they occur, and their results. Implicit data conversion can occur under the following circumstances:

1. Type conversion from one data type to another data type may only occur across the assignment operator.
2. Arithmetic conversions of precision or scale of arithmetic values may occur within an expression or across the assignment operator.

Data conversion can also occur when explicitly requested through the use of a conversion built-in function.

The target of a conversion is the field to which the converted value is assigned. In the case of a direct assignment, such as $A = B;$, in which conversion must take place, the variable to the left of the assignment operator (in this case, A) is the target.

A conversion always involves a source data item and a target data item, that is, the original representation of the value and the converted representation of the value. All of the attributes of both the source data item and the target data item are known, or assumed, at compile time.

ARITHMETIC CONVERSION

Arithmetic conversion consists of a change of scale or precision and may occur under two conditions:

1. across an assignment operator
2. automatically in an arithmetic or relational expression.

Across the assignment operator, all arithmetic data conversions are possible, that is, the scale may change between FIXED and FLOAT or vice versa and the precision may change between long and short or vice versa. When the result of a conversion from FLOAT to FIXED exceeds the range of values that can be represented by FIXED data, the result is undefined.

When the conversion takes place in the evaluation of an expression, the conversion is in one direction, i.e., FIXED to FLOAT and short to long. The results of conversion are shown in the following table:

SOURCE

TARGET	FIXED SHORT	FIXED LONG	FLOAT SHORT	FLOAT LONG
FIXED SHORT	no change	truncation on most significant	truncation on least significant	truncation on least significant
FIXED LONG	no loss of significance	no change	truncation on least significant	truncation on least significant
FLOAT SHORT	no loss of significance	truncation on least significant	no change	truncation on least significant
FLOAT LONG	no loss of significance	no loss of significance	no loss of significance	no change

Results of Arithmetic Operations

The following rules define the attributes of the results of the arithmetic operations:

1. Scale: Prefix operations yield the same scale as the operand. Infix +, -, and * operators produce a FIXED result if both operands are FIXED. The scale of all other infix operations is FLOAT.
2. Precision: The resulting precision of any arithmetic operation is the largest precision of the operands.

31 MARCH 1972

Some special cases of the exponentiation operation are defined as follows for the expression $A**B$:

1. If $A = 0$ and $B > 0$, the result is 0.
2. If $A = 0$ and $B < 0$, the ERROR condition is raised.
3. If $A = 0$ and $B = 0$, the result is 1.
4. If $A < 0$ and B is not fixed-point, the ERROR condition results.

TYPE CONVERSION

Type conversion is the process of changing the attributes of a data item (e.g., character string to bit string, ENTRY to CHARACTER) from one data type to another. This process is accomplished through the use of the assignment operation and the built-in functions. The operand on the left-side of the assignment operation is considered to be the target and the expression of the right-side as the source. The attributes of the target are determined from the declaration of the target variable.

The following table defines all the permitted type conversions in Apple. Where a conversion is permitted, a number is shown referring to one of the notes following this table. The "=" symbol indicates that the source and target data types are equivalent and no type conversion is necessary.

SOURCE DATA TYPE

TARGET DATA TYPE	Arith- metic	Charac- ter string	Bit string	Label	Locator	File	Entry
Arith- metic	1	2	4				
Charac- ter string	3	=	3			3	3
Bit string	5		=				
Label				=			
Locator	12				6 - 11		
File						=	
Entry		13					=

1. Arithmetic Conversion

See the previous section in this chapter.

2. Character-string to Arithmetic

The conversion from character-string data items to arithmetic data items is accomplished by means of the built-in functions FIXED and FLOAT (See Appendix 1) or through the use of the GET statement with the STRING option. This conversion is not permitted across the assignment operator.

31 MARCH 1972

3. Conversions to Character-string

The conversion of arithmetic, bit-string, file, or entry data items to a character string value may be accomplished by the CHAR built-in function. This conversion is not permitted across the assignment operator. The conversion of bit strings produces the character 0 for every 0 bit and the character 1 for every 1 bit to form a character string of the same length as the source bit string. Arithmetic values are converted to decimal arithmetic constants with possible leading minus signs. File variables are converted to the names of the corresponding files. Entry variables are converted to the names of the corresponding entry points.

4. Bit-string to Arithmetic

This conversion may occur across the assignment operator or when explicitly specified by the FIXED or FLOAT built-in functions. If the source bit-string is less than 48 bits long, it is interpreted as an unsigned binary integer with a precision equal to the length of the bit-string; the result of this conversion is a positive fixed value that may undergo further arithmetic conversion if required by the target data type. If the length of the source bit-string is longer than 47 bits, a diagnostic message is printed indicating an illegal conversion. The results of bit-string to arithmetic conversion are undefined if the length of the bit-string is unknown at compile time (e.g., if the length has been specified by the REFER option or as BIT(*) for a parameter).

5. Arithmetic to Bit-string

This conversion may occur across the assignment operator. The arithmetic data item is converted, if necessary, to FIXED scale long precision as defined under Arithmetic Conversions. This is treated as a bit-string of length 64 and assigned to the target bit-string in accordance with the normal rules for bit-string assignment.

6. Offset to Pointer

An offset variable is converted to a pointer value across the assignment operator or by the use of the POINTER

31 MARCH 1972

built-in function. The relative offset value is combined with the associated file origin to produce an absolute pointer value.

7. Pointer to Offset

A pointer value is converted to an offset value across the assignment operator or by the use of the OFFSET built-in function. The resulting offset value represents the relative difference between the actual pointer value and the associated file origin. The result is undefined if the pointer does not identify a generation of data in the file specified in the declaration of the offset.

8. Descriptor to Pointer

This conversion occurs across the assignment operator. The pointer value is extracted from the descriptor data item. The length value of the descriptor is ignored.

9. Pointer to Descriptor

When conversion occurs across the assignment operator, the source pointer value is interpreted as a descriptor value that indicates a length of zero. A length value may be included by using the DESCR built-in function (see Appendix 1).

10. Offset to Descriptor

An offset variable is converted to a descriptor value across the assignment operator. The relative offset value is combined with the associated file origin to form a descriptor value that indicates a length of zero.

11. Descriptor to Offset

A descriptor variable is converted to an offset value across the assignment operator. The resulting offset value is the relative difference between the pointer value in the descri-

31 MARCH 1972

ptor and the associated file origin. The result is undefined if the descriptor does not identify a generation of data in the file specified in the declaration of the offset.

12. Arithmetic to Locator

Any attempt to assign or convert an arithmetic value to a pointer or offset variable results in a diagnostic message indicating illegal conversion. However, the descriptor variable has dual attributes (both arithmetic and locator data types) and can be assigned arithmetic values with no conversion required with the exception of arithmetic conversion to long precision.

13. Character-string to Entry Value

Conversion of a character string value to an entry value is accomplished through the ENTRY built-in function (see Appendix 1). The source character-string value must be a legitimate name of an external procedure that is known to the execution environment. If the procedure can be located, the address of its entry point is returned by the ENTRY function as the entry value.

CHAPTER 5 -- DATA DESCRIPTIONINTRODUCTION

An identifier appearing in an Apple program may refer to one of many classes of objects. For, example, it may represent a variable referring to a floating-point number; it may refer to a file; it may be a variable referring to a pointer or offset; etc.

The recognition of an identifier as a particular name is established through the declaration of the name. The declaration provides a means for associating properties with a name. These properties and the scope of the name itself together make up the data attributes of an identifier.

When an identifier is used in a given context in a program, attributes must be known in order to assign a unique meaning to the occurrence of the identifier. For example, if an identifier is used to represent an arithmetic variable, the scale, precision, and storage class must be known. Examples of attributes are:

CHARACTER(50) -- Association of this attribute with an identifier defines the identifier as representing a variable referring to a string of 50 characters.

FLOAT -- Association of this attribute with an identifier defines the identifier as representing a variable referring to arithmetic data.

EXTERNAL -- Association of this attribute with an identifier defines the identifier as a name with a global scope.

DECLARATIONS

A given identifier is established as a name which holds throughout a certain scope in the program (see "Scope of Declarations" in this chapter) and a set of attributes may be associated with the name by means of a declaration.

If a declaration is made in a block, then the name is said to be internal to that block and contained blocks unless redeclared. However, a given identifier may be established

31 MARCH 1972

in different parts of a program as referring to separate objects. For example, an identifier may represent an arithmetic variable in one part of a program and an entry constant in another part. These two parts cannot overlap. Each separate use of the identifier is established by means of a separate DECLARE statement. The rules of scope distinguish between references to different uses of the identifier.

EXPLICIT DECLARATIONS

Explicit declarations are made through the use of the DECLARE statement (see Chapter 8), label prefixes and specification in a parameter list. By this means, an identifier can be established as a name and can be given a certain set of attributes.

Only one DECLARE statement can be used to establish an internal name. However, in the case of a parameter, a complementary explicit declaration is required. The appearance of the identifier in the parameter list specifies that the identifier has the parameter attribute. This must be combined with an explicit declaration in a DECLARE statement to provide other data attributes. These multiple declarations of the same name must be internal to the same block. This is known as a complementary set of declarations.

Two or more declarations of the same identifier, internal to the same block, constitute a multiple declaration of that identifier only if they have identical qualification (including the case of two or more declarations of an identifier at level 1, i.e., scalars or major structures). Multiple declarations are in error.

Example:

```

DECLARE 1 A,
        2 B,
        2 B,
        2 C,
        3 D,
        2 D;

```

B has a multiple declaration.

31 MARCH 1972

Label Prefixes

The use of an identifier as a label prefix to a PROCEDURE or ENTRY statement causes an explicit declaration of that identifier as a name with the following attributes:

ENTRY with no returns attributes

EXTERNAL if the entry point belongs to an external procedure

INTERNAL if the entry point belongs to an internal procedure

If the PROCEDURE or ENTRY statement applies to the entry point of an internal procedure, the declaration of the identifier occurs in the block that immediately contains the internal procedure. If the entry point belongs to an external procedure, the declaration occurs in an imaginary block of which the sole contents are the external procedure concerned and the set of declarations generated for its entry points.

A label acting as a prefix to any other statement is an explicit declaration of the identifier as a statement label constant. The declaration occurs within the block containing the prefix.

Parameters

An identifier that appears in a parameter list of an ENTRY or PROCEDURE statement is explicitly declared as a name with the attribute "parameter". Further attributes must be supplied by the programmer in a DECLARE statement.

CONTEXTUAL DECLARATIONS

The syntax of Apple allows the contextual declaration of built-in functions. Such contextual declarations will not, however, override any explicit declaration of the same identifier whose scope includes the block containing a statement that might otherwise cause contextual declaration.

31 MARCH 1972

An undeclared identifier is contextually declared with the attribute BUILTIN if it appears in either of the contexts:

- a. It appears anywhere that it is legal for a function or pseudo-variable to appear and is followed by an argument list.
- b. It follows the keyword CALL in a CALL statement.

A contextual declaration is treated as if it had been made in the external procedure, even if the reference is made in an internal block. The scope of a contextually declared name is the entire external procedure, except for any internal blocks in which the same identifier is explicitly declared. Explicit declarations have priority over contextual declarations.

SCOPE OF DECLARATIONS

When a declaration of an identifier is made in a block, there is a certain well-defined region of the program (see "Block Structure" in Chapter 2) over which the declaration is applicable. This region is called the scope of the declaration.

The scope of a declaration of an identifier is defined as that block B to which the declaration is internal, but excluding from the block B all contained blocks to which another declaration of the same identifier is internal. Block B may be the imaginary block that is considered to contain the declaration of external entry constants, as discussed under "Label Prefixes" in this chapter.

A name is said to be known only within its scope. This definition suggests a basic rule on the use of names:

All appearances of an identifier that are intended to represent a given name in a program must lie within the scope of that name.

The above rule has many implications. One of the most important is the limitation of transfer of control by the statement GO TO A; where A is a statement label constant.

The statement GO TO A;, internal to a block B, can cause a transfer of control to another statement internal to block B or to a statement in a block containing B, and to no other statement. In particular, it cannot transfer control to any point within a block contained in B.

31 MARCH 1972

In general, distinct declarations of the same identifier imply distinct names with distinct non-overlapping scopes. It is possible, however, to establish the same name for distinct declarations of the same identifier by means of the EXTERNAL attribute. The EXTERNAL attribute is defined as follows:

A declaration of an identifier that specifies the identifier as EXTERNAL is called an external declaration for the identifier. All external declarations for the same identifier in a program will be linked and considered as establishing the same name. The scope of this name will be the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must agree since all the declarations involve a single name and refer to the same object.

The EXTERNAL attribute can be used to communicate between different external procedures or to obtain non-continuous scopes for a name within an external procedure.

An external name is a name that has the scope attribute EXTERNAL. If a name is not external, it is said to be an internal name and has the scope attribute INTERNAL.

The following examples illustrate scope of declarations. The numbers on the left are for reference only and are not part of the procedure.

31 MARCH 1972

Example 1.

```

1.  A: PROCEDURE;
2.      DECLARE (X, Z) FLOAT;
      ...
3.      B: PROCEDURE(Y);
4.          DECLARE Y BIT(6);
5.          C: BEGIN;
6.              DECLARE (A, X) FIXED;
              ...
7.              Y: RETURN;
              END C;
          END B;
8.      D: PROCEDURE;
9.          DECLARE X CHARACTER(20)
              EXTERNAL;
          ...
          END D;

      END A;
    
```

Since entry names of external procedures have the attribute EXTERNAL, the scope of the entry name A and of the character string X declared in line 9 above may include parts of other external procedures of the program. The following table gives an explanation of the scope and use of each name:

Line	Name	Uses	Scope (by block name)
1	A	external entry name	all of A except C
2	X	floating-point variable	all of A except C & D
2	Z	floating-point variable	all of A
3	B	internal entry name	all of A
4	Y	bit string	all of B except C
5	C	statement label	all of B
6	A	fixed-point variable	all of C
6	X	fixed-point variable	all of C
7	Y	statement label	all of C
8	D	internal entry name	all of A
9	X	character string	all of D

31 MARCH 1972

Example 2.

```

1.   A: PROCEDURE;
      DECLARE X EXTERNAL FLOAT;
      ...
2.   B: PROCEDURE;
      DECLARE X FIXED;
      ...
3.   C: BEGIN;
      DECLARE X EXTERNAL FLOAT;
      ...
      END C;
      END B;
      END A;
4.   D: PROCEDURE;
      DECLARE X FIXED;
      ...
5.   E: PROCEDURE;
      DECLARE X EXTERNAL FLOAT;
      ...
      END E;
      END D;

```

In example 2, there are five separate declarations for the identifier X. Declaration 2 declares X as a fixed-point variable name; its scope is all of block B except block C. Declaration 4 declares X as another fixed-point variable name, distinct from that of declaration 2; its scope is all of block D except block E.

Declarations 1, 3, and 5 all establish X as a single external program. Declarations 2 and 4 establish X as a FIXED scalar in blocks B and D.

31 MARCH 1972

DEFAULT ATTRIBUTES

Some attributes are given to identifiers by explicit and contextual declarations. Generally these do not constitute the full set of attributes and the remaining attributes are deduced according to the following set of default rules:

1. In the absence of contradictory specification, the following attributes may be deduced from those already specified:

Specified	Defaults
AUTOMATIC	INTERNAL
BIT	VARIABLE
BUILTIN	CONSTANT, INTERNAL
BASED	INTERNAL
CHARACTER	fixed-length
CONDITION	CONSTANT, INTERNAL
CONSTANT	STATIC, INTERNAL
DESCRIPTOR	VARIABLE
ENTITY	INTERNAL
ENTRY	EXTERNAL, CONSTANT
EVENT	INTERNAL, CONSTANT
EXTERNAL	STATIC, VARIABLE
FILE	INTERNAL
FILE_SET	INTERNAL
FIXED	BINARY(47), VARIABLE
FLOAT	DECIMAL(14), VARIABLE
INITIAL	VARIABLE
INTERNAL	AUTOMATIC
LABEL	VARIABLE
OFFSET	VARIABLE
POINTER	VARIABLE
SET	INTERNAL
REGISTER	INTERNAL
STATIC	INTERNAL
VARIABLE	INTERNAL

31 MARCH 1972

2. For all identifiers that are scalars, elements of a structure, or arrays of non-structured elements, one of the following attributes must be specified in a DECLARE statement:

BIT (length-specification)
BUILTIN
CHARACTER (length-specification)
CONDITION
DESCRIPTOR
ENTITY
ENTRY
EVENT
FILE
FILE_SET
FIXED
FLOAT
LABEL
OFFSET
POINTER
SET

LIST OF ATTRIBUTES

Following is a detailed description of the attributes that can appear in a DECLARE statement. Alternative attributes are discussed together.

31 MARCH 1972

AUTOMATIC, STATIC, REGISTER, and BASED

The storage class attributes are used to specify the type of storage allocation to be used for level one data variables.

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block. A data value may be represented by an automatic variable only as long as the block to which that variable is internal remains active. The value is lost upon exit from the block.

STATIC specifies that storage is to be allocated when the procedure containing the declaration is first invoked and is not released until program execution has been completed.

REGISTER specifies that storage is to be allocated within the STAR hardware registers whenever the declaring block is activated in the same manner that automatic variables are allocated. The storage is released and the values are lost upon exit from the block. This storage class is the most efficient from the point of view of access; however, it has the most restrictions.

BASED specifies that full control of allocation will be maintained by the programmer through the use of the **ALLOCATE** and **FREE** statements. A variable with the **BASED** attribute is allocated storage only upon the execution of an **ALLOCATE** statement specifying that variable. This allocation remains even after termination of the block in which it was allocated. The storage will remain allocated for that variable until the execution of a **FREE** statement which specifies that variable. All current allocations of based variables are available at any time. Unique reference to a particular allocation is provided by a locator value qualifying the based reference. A based variable can also be used to reference data of any storage class by associating the based variable name with a locator qualifier that identifies that data. Based storage is the most powerful of the Apple storage classes, but it must be used carefully. Many of the safeguards against error that are provided for other storage classes cannot be provided for based.

31 MARCH 1972

General format:

storage-class-attribute ::=

$$\left\{ \begin{array}{l} \text{STATIC} \\ \text{AUTOMATIC} \\ \text{REGISTER [(register-specification)]} \\ \text{BASED [(locator-variable)]} \end{array} \right\}$$

General rules:

1. Automatic, register and based variables can have internal scope only. Static variables may have either internal or external scope.
2. Storage class attributes cannot be specified for conditions, entities, entries, built-in functions, events, or members of structures.
3. The storage class attributes STATIC, AUTOMATIC and BASED cannot be specified for parameters.
4. Variables declared with adjustable array bounds or string lengths may only have the BASED storage class attribute.
5. For a structure variable, a storage class attribute can be given only for the major structure name. The attribute then applies to all elements of the structure or to the entire array of structures. Storage is always allocated for a complete major structure. The contained items may not be independently allocated or freed.
6. If, during the evaluation of an expression, a based variable is allocated or freed, the result of the expression is undefined if the variable is used elsewhere in the statement.
7. The following rules govern the use of based variables:
 - a. The locator variable named in the BASED attribute must be an unsubscripted scalar locator variable. This restriction does not apply to explicit locator qualifiers, which may be general locator expressions.

31 MARCH 1972

- b. If no locator variable is named in the BASED attribute, any reference to the based variable must have an explicit qualifier. This does not apply to a based variable that is the object of a REFER option or that is to be allocated through the use of an ALLOCATE statement.
- c. A reference to a based variable without an explicit locator qualifier is implicitly qualified by the locator variable named in the BASED attribute specification in the DECLARE statement for the based variable. Identifiers in this implicit qualification are those known in the declaring block.

Example:

```

DECLARE P POINTER,
          B BASED(P);
...
BEGIN;
    DECLARE P POINTER;
    ...
L:  B = X;

```

In the statement labelled L, the assignment $B = X$; has the same effect as:

```
P->B = X;
```

where P is the name known in the outer block, not the one declared in the begin block.

For the results of a reference to be defined:

- i. The attributes of the based variable must be the same as those of the data identified by the locator qualifier.
- ii. The declared maximum length of a string with the attributes BASED VARYING must be equal to the maximum length of the string identified by the locator qualifier used in the reference.

31 MARCH 1972

- iii. The length of a fixed length string with the attribute BASED should be equal to the length of the string identified by the locator qualifier used in the reference.
 - iv. The aggregate type and data type of all elements of the structure must agree up to and including all of the level-2 items that contain the referenced sub-item. A level-2 item is an immediately contained member of structure, i.e., is not contained in any other member.
- d. When a reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated locator variable identifies the generation of data. If the reference is to a component of a based structure, a second temporary locator variable is created to determine the location of the component in relation to the beginning of the structure.
- e. Array bounds and string lengths of identifiers declared with the BASED attribute are evaluated dynamically with each reference to the based variable. Therefore, the asterisk notation for dimensions and lengths is not permitted. A reference to a component of a based structure causes evaluation of sufficient elements of the structure to determine the position of the component.

Example:

```

DECLARE P POINTER,
        M FIXED,
        N FIXED,
        A(2*M, 2*(M+N)) FLOAT
        BASED(P);

```

At every reference to an element of A, variables M and N must contain

31 MARCH 1972

values that correspond to the dimensions of the generation of A being accessed.

- f. When a based variable is allocated using the ALLOCATE statement, expressions for bounds and lengths are evaluated at the time of allocation in the environment of the declaration.
- g. The REFER option can be used to create structures that define their own adjustable bounds and lengths, i.e., self-defining data. The REFER option may be used in a DECLARE statement to define a bound of an array or the length of a string.

General format:

refer-option ::=

expression REFER (scalar-name)

where "scalar-name" is a reference, possibly qualified, but not subscripted or locator qualified. The reference must be to a scalar item preceding the REFER option in the structure.

The REFER option can not be used in the declaration of a structure which is named in the LIKE attribute for another identifier. (See "LIKE" later in this chapter for details.)

31 MARCH 1972

Example:

```

DECLARE P POINTER,
        M FIXED,
        N FIXED,
        1 A BASED(P),
        2 N1 FIXED,
        2 N2 FIXED,
        2 N3 FIXED,
        2 B(M+3 REFER(N1),
           M*N REFER(N2)) FLOAT,
        2 C CHAR(2*M*N REFER(N3)),
        2 D FIXED;

M = 5;
N = 10;
ALLOCATE A;

```

This will cause space to be allocated for A with the bounds of B, 8 and 50, and the length of C, 100. N1, N2 and N3 will be set to 8, 50 and 100 respectively. A reference to D will cause expressions involving N1, N2 and N3 to be evaluated.

- h. The INITIAL attribute may be specified for a based variable. The values are used only upon explicit allocation of the based variable with an ALLOCATE statement.
 - i. Whenever a based variable containing arrays or strings is passed as an argument, the bounds and lengths are determined at the time the argument is passed and remain fixed throughout execution of the invoked block.
8. The following rules govern the use of register variables:
- a. If a scalar arithmetic variable with the REGISTER attribute is passed as an argument, its contents are passed by value rather than by reference as is done for all other storage classes (see "Correspondence of Argument and Parameters" in Chapter 2).
 - b. Although a variable with the REGISTER attribute may be used as the argument to

31 MARCH 1972

the ADDR built-in function, the pointer value returned by the function is undefined outside the block in which it was evaluated.

- c. The "register-specification" must be an unsigned integer constant in the range 0 - 255. If a register specification is given and the use of the register conflicts with the standard use of that numbered register, the results of the procedure are undefined. REGISTER variables declared with a register-specification will not have their values preserved across calls.
- d. If no register specification is given, a register number will be assigned by the compiler. These values will be preserved across calls.

31 MARCH 1972

BINARY and DECIMAL

The precision attribute is used to specify the minimum number of significant digits to be maintained for the storage of arithmetic data variables. The precision attribute can be specified in terms of either binary or decimal digits as indicated by the BINARY or DECIMAL qualifier.

General format:

$$\text{precision-attribute} ::= \left\{ \begin{array}{l} \text{BINARY} \\ \text{DECIMAL} \end{array} \right\} [(\text{number-of-digits} [,0])]$$

The "number-of-digits" is an unsigned non-zero decimal integer constant.

General rules:

1. The "number-of-digits" specifies the minimum number of digits to be maintained for data items assigned to the variable. The number of digits is specified for both fixed-point and floating-point variables.
2. An optional scale factor of zero may be specified for fixed-point variables only.
3. The maximum precision that is supported is:

BINARY --- 47 bits
DECIMAL --- 14 decimal digits

If the "number-of-digits" specified exceeds these limits the maximum value will be used.

4. The actual precision, p , that will be used is determined from the "number-of-digits", d , as follows:

If the precision-attribute is BINARY then
 if $1 \leq d \leq 23$ then $p = 23$.
 if $24 \leq d \leq 47$ then $p = 47$.

If the precision-attribute is DECIMAL then
 if $1 \leq d \leq 6$ then $p = 6$.
 if $7 \leq d \leq 14$ then $p = 14$.
5. If the "number-of-digits" is omitted, the maximum precision is assumed.

31 MARCH 1972

BIT and CHARACTER

The BIT and CHARACTER attributes are used to specify string variables. The length of the string is defined in terms of the number of elements to be maintained, where an element is either a bit or character.

General format:

```
string-attribute ::=
    { BIT
      CHARACTER } ( length-specification ) [ VARYING ]
```

General rules:

1. The VARYING attribute specifies that the maximum length of the string has been specified by the length-specification. The current length at any time is the length of the current value of the string. VARYING may only be applied to character strings.
2. The declared attributes (including length and VARYING) of a string with the attribute BASED must match the attributes of the string identified by a locator variable in a reference.
3. The length-specification must immediately follow the CHARACTER or BIT attribute at the same factoring level.
4. In the case of a parameter, the length may be specified by an asterisk. This indicates that the length of the string is determined from the corresponding argument string being passed.
5. The length-specification of strings declared with the AUTOMATIC or STATIC attributes must be an unsigned integer constant.
6. The length-specification of a BASED variable string may be declared using the REFER option (see the ALLOCATE statement).
7. The current length of an uninitialized varying-length string is undefined before assignment.

31 MARCH 1972

BUILTIN

The BUILTIN attribute specifies that any reference to the associated name within the scope of the declaration is to be interpreted as a reference to the built-in function or pseudo-variable of the same name. The built-in functions and pseudo-variables of Apple are listed in Appendix 1.

General format:

```
built-in-attribute ::= BUILTIN
```

General rules:

1. BUILTIN is used to refer to a built-in function or pseudo-variable in a block within a containing block in which the same identifier has been declared to have another meaning.

Example:

```
A:
PROCEDURE;
    DECLARE SQRT ENTRY (FLOAT)
        RETURNS (FLOAT);
    ...
    X = SQRT(Y); /*This calls the external
                procedure SQRT */
BEGIN;
    DECLARE SQRT BUILTIN;
    X = SQRT(Y); /* This calls the
                built-in function SQRT */
END;
END;
```

2. If the BUILTIN attribute is declared for an entry constant, there can be no other explicitly declared attributes for the entry constant except INTERNAL.
3. The BUILTIN attribute cannot be declared for parameters.
4. The BUILTIN attribute must be specified for any parameterless built-in functions or pseudo-variables that are referenced by the program (e.g., NULL, TIME, ONFILE).

31 MARCH 1972

CHARACTER

See BIT and CHARACTER

CONSTANT

The CONSTANT attribute specifies that the associated identifier is the name of a constant (a value which cannot change during program execution).

General format:

constant-attribute ::= CONSTANT(value-list)

where the specification of "value-list" is given in the section on the INITIAL attribute in this chapter.

General rules:

1. The CONSTANT attribute may only be specified for level-1 identifiers with arithmetic, string, locator, or LABEL attributes. It can not be specified for parameters, structures, or any variables.
2. Only one constant value may be specified for a scalar identifier. A list of values can be specified for a constant array; however, the number of values must match the number of elements in the array.
3. The same rules apply to the "value-list" as apply to the initial-value-list described in the section on the INITIAL attribute with the exception that the asterisk notation (used to skip or ignore elements) is not permitted.
4. The only storage class attribute that may be specified for an identifier with the CONSTANT attribute is STATIC.
5. The only scope attribute that may be specified for an identifier with the CONSTANT attribute is INTERNAL.
6. The values of LABEL constants must be label prefixes within the block of the declaration.

31 MARCH 1972

CONDITION and EVENT

The EVENT or CONDITION attribute specifies that the identifier refers to an interrupt.

General format:

$$\text{interrupt-attribute} ::= \left\{ \begin{array}{l} \text{EVENT} \\ \text{CONDITION} \end{array} \right\}$$

General rules:

1. No attributes other than scope (INTERNAL or EXTERNAL) can be specified for interrupt identifiers. These identifiers can not be declared as arrays or members of structures.
2. Only user defined conditions can be declared. System defined conditions (see Appendix 2) are treated as keywords in the ON, REVERT, and SIGNAL statements.

DECIMAL

See BINARY and DECIMAL

DESCRIPTOR

See OFFSET, POINTER, and DESCRIPTOR.

31 MARCH 1972

Dimension

The dimension attribute specifies the number of dimensions of an array and the bounds of each dimension. The dimension attribute specifies the bounds (only the upper bound or both the upper and lower bounds) or indicates, by the use of an asterisk, that the actual bounds for the array are to be taken from the passed parameter.

General format:

```
dimension-attribute ::= (bound [, bound] ...)
```

```
bound ::= { [ lower-bound : ] upper-bound } | *
```

where "upper-bound" and "lower-bound" are fixed scalar expressions.

General rules:

1. The number of "bounds" specified indicates the number of dimensions in the array unless the variable being declared is contained in an array of structures. In this case, additional dimensions are also inherited from the containing structure.
2. The bound specification indicates the bounds as follows:
 - a. If only the upper bound is given, the lower bound is assumed to be 1.
 - b. The value of the fixed scalar expression is evaluated on allocation of storage and on reference; the value of the lower bound must be less than or equal to the value of the upper bound.
 - c. An asterisk used as a bound specification indicates that the actual bounds of an array parameter are to be the bounds of its associated array argument.
3. Bounds that are expressions are known as adjustable bounds and are evaluated when storage is allocated for the array and when the array is referenced. For parameters, bounds can be only asterisks or optionally signed integer constants.

31 MARCH 1972

4. The bounds of arrays declared with the attributes AUTOMATIC, REGISTER, or STATIC must be optionally signed integer constants.
5. The dimension attribute must be the first attribute to follow the array name (or parenthesized list of names if the dimension attribute is being factored) in the declaration. Intervening blanks are optional.
6. The REFER option can be used to specify the bounds of a BASED variable (see the ALLOCATE statement).
7. The total number of elements in an array may not exceed 65535.

ENTITY

The ENTITY attribute specifies a variable that may be manipulated by the INSERT, REMOVE, FIND, and FOR EACH statements.

General format:

```
entity-attribute ::= ENTITY [ (locator-variable) ]
```

General rules:

1. Specification of the ENTITY attribute implies that the named identifier is a structured based variable. A system function will be provided to record in a file the structure declaration for each entity. At compile time this structure declaration will replace the entity declaration. It is intended that standard entity declarations for a project will all be recorded in the same file.
2. The ENTITY attribute may only be applied to a level-1 identifier which may have no other declared attributes.
3. The length of an identifier given the ENTITY attribute can not exceed 8 characters.

31 MARCH 1972

ENTRY

The ENTRY attribute specifies that the identifier is being declared as an entry constant or entry variable. It is also used to describe the attributes of the parameters of the entry point.

General format:

```
entry-attribute ::= ENTRY [(parameter-attribute-list
                           [, parameter-attribute-list]...)]
                  [ RETURNS(attribute-list) ]
```

Rules for "parameter-attribute-lists":

1. A parameter-attribute-list describes the attributes of a single parameter; the parameter name is not given.
2. The parameter-attribute-lists must appear in the same order as the parameters they describe.
3. The attributes describing a scalar parameter may appear in any order within the parameter-attribute-list. The attributes within a list must be separated by blanks; lists must be separated by commas. For an array parameter, the dimension attribute must be the first specified.
4. Array bounds and string lengths may only be specified by decimal integer constants or by asterisks.
5. Parameter-attribute-lists may not contain the attributes STATIC, BASED, LIKE, AUTOMATIC, BUILTIN, EXTERNAL, INTERNAL, CONSTANT, or INITIAL.

General rules:

1. The ENTRY attribute may not be specified:
 - a. for an array or within a structure,
 - b. within a RETURNS attribute, or
 - c. with the BUILTIN attribute.
2. The factoring of attributes is not permitted within the set of parameter-attribute-lists of an ENTRY attribute specification.

31 MARCH 1972

3. An external entry constant must be given the attribute ENTRY, otherwise it is contextually declared with the BUILTIN attribute and is treated as the name of a built-in function.
4. All entry names which are invoked as functions in the procedure must be declared with a RETURNS attribute.
5. The appearance of an identifier as a label prefix of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry constant, thus the same identifier may not be declared in a DECLARE statement in the same block.
6. The attribute INITIAL may not be specified for entry variables.
7. The attribute INTERNAL may not be specified for entry constants.
8. An identifier declared with the ENTRY attribute is assumed to be an entry constant, unless the VARIABLE attribute is also specified.

EVENT

See CONDITION and EVENT

31 MARCH 1972

EXTERNAL and INTERNAL

The EXTERNAL and INTERNAL attributes specify the scope of a name. INTERNAL specifies that the name can be known only in the declaring block and its contained blocks. EXTERNAL specifies that the name may be known in other blocks containing an external declaration of the same name.

General format:

$$\text{scope-attribute} ::= \left\{ \begin{array}{l} \text{EXTERNAL} \\ \text{INTERNAL} \end{array} \right\}$$

General rules:

1. The lengths of identifiers given the EXTERNAL attribute cannot exceed 8 characters.
2. The lengths of identifiers given the INTERNAL attribute cannot exceed 31 characters.
3. The scope attributes can only be applied to level-1 identifiers.

FILE

The FILE attribute specifies that the identifier being declared is a file variable.

General format:

file-attribute ::= FILE VARIABLE

General rules:

1. Only the following attributes may be specified with the file-attribute:

Scope attributes:	INTERNAL EXTERNAL
Storage class attributes:	AUTOMATIC STATIC REGISTER BASED
Dimension attribute	

31 MARCH 1972

2. The RETURNS attribute in an entry declaration or the RETURNS option in a PROCEDURE or ENTRY statement may specify the FILE attribute if the corresponding procedure returns a file value.
3. File variables may be used in the following contexts:
 - a. as arguments to functions and procedures,
 - b. as arguments to an I/O condition name in SIGNAL, REVERT, or ON statements,
 - c. as arguments to a FILE option in a GET or PUT statement,
 - d. in the assignment of one file variable to another file variable,
 - e. as operands of the = and <= comparison operators (two file-variables compare equal only if they represent the same file value),
 - f. in the declaration of an OFFSET variable,
 - g. in the INSERT, REMOVE, FIND, FOR EACH, and LET statements.
4. On-units can be established for files whose identity is represented by a file-variable.

Example:

```

DECLARE F FILE VARIABLE,
        G FILE VARIABLE;
/* Request that MCTS File System open a file
   and set the file variable F */
G = F;
...
L1: ON ENDFILE(G);
L2: ON ENDFILE(F);
/* Statements labeled L1 and L2
   have identical effect. */

```

FILE SET

See SET attribute

31 MARCH 1972

FIXED and FLOAT

The `FIXED` and `FLOAT` attributes specify the scale of the arithmetic variable being declared. `FIXED` specifies that the variable is to represent fixed-point data items. `FLOAT` specifies that the variable is to represent floating-point data items.

General format:

$$\text{scale-attribute} ::= \left\{ \begin{array}{l} \text{FIXED} \\ \text{FLOAT} \end{array} \right\}$$

General rule:

1. Fixed-point data items only represent integer values. No fractional digits can be represented by a fixed-point variable.

Assumptions:

1. If only the scale attribute `FIXED` is specified, the precision attribute `BINARY (47)` is assumed.
2. If only the scale attribute `FLOAT` is specified, the precision attribute `DECIMAL(14)` is assumed.

INITIAL

The `INITIAL` attribute specifies an initial constant value to be assigned to a data item whenever storage is allocated for the variable.

31 MARCH 1972

General format:

```
initial-attribute ::= INITIAL(value-list)
```

```
value-list ::= item[, item] ...
```

```
item ::= {
  constant
  *
  iteration-specification
}
```

```
iteration-specification ::=
```

```
{
  (iteration-factor) constant
  (iteration-factor) *
  (iteration-factor) (item[, item] ...)
}
```

General rules:

1. The INITIAL attribute may only be assigned to level-1 variables with arithmetic, string, locator, or label attributes; it cannot be given for parameters, structures, entry variables, or file variables.
2. In the following rules, the term "constant" denotes one of the following:
 - arithmetic-constant
 - character-string-constant
 - bit-string-constant
 - statement-label constant
 - the value of the NULL built-in function
3. Only one constant value can be specified for a scalar variable; a list of values can be specified for an array variable.
4. Constant values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly).
5. If too many constant values are specified for an array, excess values are ignored; if not enough are specified, the remainder of the array is not initialized.
6. Each item in the list can be a constant, an asterisk denoting no initialization for a particular element, or an iteration specification.

31 MARCH 1972

7. The "iteration-factor" specifies the number of times the constant, item list, or asterisk is to be repeated in the initialization of elements of an array. If a constant follows the "iteration-factor", then the specified number of elements are to be initialized with that value. If a list of items follows the "iteration-factor", then the list is to be repeated the specified number of times, with each item initializing an element of the array. If an asterisk follows the "iteration-factor", the specified number of elements are to be skipped in the initialization operation.
8. The "iteration-factor" must be an unsigned decimal integer constant.
9. A based array with adjustable bounds or a based string variable with an adjustable length cannot be initialized.
10. For the initialization of a string array, both an "iteration-factor" and a "string-repetition-factor" may be used. If only one parenthesized integer precedes the string constant, it is assumed to be the "string-repetition-factor" of the initial value for a single element of the array. Consequently, to cause initialization of more than one element of a string array, both the iteration factor and the string repetition factor, in that order, must be stated explicitly, even if the string repetition factor is (1).

Example:

((2)'A') is equivalent to ('AA') (for a single element)

((2)(1)'A') is equivalent to ('A', 'A') (for two elements)

11. Label constants given as initial values for label variables must be contained within the block in which the label variable declarations occur. The INITIAL attribute may only be specified for label variables of the AUTOMATIC storage class.
12. The only initial value that can be specified for a locator variable is the value of the NULL built-in function. This is the only function reference that may appear in an initialization list.

31 MARCH 1972

LABEL

The LABEL attribute specifies that the identifier being declared can have statement labels as values.

General format:

label-attribute ::= LABEL

General rules:

1. If the label variable is a parameter, its value can be any statement label variable or constant passed as an argument by the caller.
2. An entry name cannot be a value of a label variable.
3. The INITIAL attribute cannot be specified for label variables with the STATIC, REGISTER, or BASED storage class.
4. The CONSTANT attribute may be declared for identifiers with the LABEL attribute.

31 MARCH 1972

LIKE

The LIKE attribute specifies that the name being declared is a structure variable with the same structuring as that for the name following the attribute keyword LIKE. Substructure names, element names, and attributes for substructure names and element names are to be identical.

General format:

like-attribute ::= LIKE identifier

General rules:

1. The "identifier" must be the unsubscripted name of a level-1 variable.
2. The "identifier" must be known in the block containing the LIKE attribute specification. Neither the "identifier" nor any of its substructures may be declared with the LIKE attribute or the REFER attribute. (Appendix 8 shows a method for declaring similar structures with the REFER option.)
3. Attributes of the level-1 identifier itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the "identifier" following the keyword LIKE represents an array of structures, its dimension attribute is not carried over. The attributes of substructure names and element names, however, are carried over; if the attributes that are carried over contain names, these names are interpreted in the block containing the LIKE attribute.
4. If a direct application of the description to the structure declared with the LIKE attribute would cause an incorrect continuation of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1) the level numbers are modified by the addition of a constant before application.
5. Any level number following the "identifier" must be less than or equal to the level number of the variable being declared with the LIKE attribute; thus, no additional substructures or elements may be added to the created structure.

31 MARCH 1972

OFFSET, POINTER, and DESCRIPTOR

The OFFSET, POINTER, and DESCRIPTOR attributes specify locator variables. A locator variable can be used in a based variable reference to identify a particular generation of the based variable. Offset variables identify a location relative to the origin of a file; pointer variables identify any location; descriptor variables identify both the location and the length. Offset values retain their validity between jobs; this is not the case with pointer and descriptor variables.

General format:

$$\text{locator-attribute} ::= \left\{ \begin{array}{l} \text{POINTER} \\ \text{OFFSET (file-variable)} \\ \text{DESCRIPTOR} \end{array} \right\}$$

General rules:

1. The pointer or descriptor value of a locator variable or function uniquely identifies a generation of a based variable. This generation may be accessed by using the variable or function as the locator qualifier in the reference to a based variable whose evaluated attributes match those of the generation. A value of pointer type may be obtained from the built-in functions ADDR, NULL, and POINTER. A descriptor value may be obtained from the built-in function DESCR.
2. The value of a descriptor variable is used to describe the location and length of a based string variable or the location and dimension of a based vector variable. (A vector variable is defined to be a one-dimensional array of arithmetic, locator, or file variables.) When a descriptor variable is used as a locator qualifier to reference a based string or vector, the length of the string or the extent of the vector is specified by the descriptor value; all other attributes are specified by the based variable.
3. The value of an offset variable or function identifies the position of a generation of a based variable within a file relative to the origin of the file. This value may be converted to a pointer to the generation of the based variable by supplying the file and the offset value as arguments to the POINTER built-in function. A value

31 MARCH 1972

of offset type may be obtained from the built-in function OFFSET.

4. The value of a locator variable can be set in any of the following ways:
 - a. By assigning the value of the NULL built-in function.
 - b. By the ALLOCATE, LET, FIND, and FOR EACH statements.
 - c. By assignment of the value of a locator variable or function.
5. The value of a descriptor variable can also be set in the same manner as an arithmetic variable.
6. Pointer and offset variables cannot be operands of any operators other than the comparison operators = and \neq . Descriptor variables may appear in relations involving any comparison operator and may also be used in arithmetic expressions as operands to the infix + and - operators.
7. Locator data, except for descriptors, cannot be converted to any other data type, but pointer can be converted to offset, and vice versa.
8. A pointer or offset variable can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the file variables named in the OFFSET attributes are ignored.

REFER

See AUTOMATIC, STATIC, REGISTER, and BASED.

31 MARCH 1972

RETURNS

The RETURNS attribute specifies the attributes of the value returned by an external entry or an entry variable when invoked as a function.

General format:

```
returns-attribute ::= RETURNS (attribute ...)
```

General rules:

1. The attributes in the parenthesized list following the keyword RETURNS must be separated by blanks (except for attributes such as precision or length that are enclosed in parentheses).
2. Only scalar arithmetic, string, locator, or file attributes can be specified.
3. All lengths specified in the attributes must be constants.
4. The RETURNS attribute must be specified in the declaration of all entry variables and external entry names invoked as functions. The attributes given in the invoking procedure must agree with the attributes specified in the PROCEDURE or ENTRY statement of the invoked function.

31 MARCH 1972

SET

The SET attribute defines the name of a data aggregate that represents an ordered set of entity variables. Sets are identified by the file or entity that contains the set and the name of the set. A uniquely named set that is contained by the file in which it resides is referred to as a FILE_SET.

General Format:

$$\text{set-attribute} ::= \left\{ \begin{array}{l} \text{SET} \\ \text{FILE_SET} \end{array} \right\}$$

General Rules:

1. A named set may be referenced by the INSERT, REMOVE, FIND, FOR EACH, and LET statements.
2. The insertion of an entity into a set will cause the set to be created if it previously did not exist. Similarly, if all members of a set are removed, the set will be automatically deleted.
3. Multiple generations of a named set may be created, one for each containing entity. However, since a FILE_SET has no containing entity, only one generation of a FILE_SET may be created in a particular file.
4. The length of a set identifier can not exceed 8 characters.
5. No other attributes may be specified with the SET or FILE_SET attribute.

31 MARCH 1972

VARIABLE

The VARIABLE attribute is used with the ENTRY attribute to establish a name as an entry-variable or, if used with the FILE attribute, to establish a name as a file-variable.

General format:

variable-attribute ::= VARIABLE

General rules:

1. The VARIABLE attribute can only appear in a declaration with the ENTRY or FILE attribute.
2. The attribute VARIABLE is supplied by default if the ENTRY attribute is declared with any one or more of the following attributes:

AUTOMATIC
BASED
STATIC
REGISTER
Parameter

VARYING

See BIT and CHARACTER.

31 MARCH 1972

CHAPTER 6: FILE HANDLINGINTRODUCTION

This chapter contains a discussion of the Apple language facilities that are available for processing data files. The chapter is divided into three logical areas:

1. File organization and access to files.
2. Format-directed data transmission to or from sequential files.
3. Associative data storage management within structured files.

The discussion concentrates on describing how various file-handling statements are used. (The syntax rules for each statement are defined in Chapter 8.)

FILES

Any collection of data that can be transmitted automatically for the programmer between the internal memory and external storage devices of the computer is called a file. A file is known by a symbolic name and may be stored on magnetic tape, direct-access disk or drum, a deck of punched cards, or a printed listing. The actual storing, retrieving, and cataloging of files is controlled and performed by the File Management System and is not described in this manual. (The reader is referred to the File Management Manual.)

The transmission of data between internal memory and external storage devices has traditionally been called input and output (I/O). The system architecture supporting the execution of Apple programs allows these I/O operations to be performed more effectively by dividing them into two-stage processes:

1. The user program reads all of its input data from virtual memory and stores all of its output results in virtual memory.
2. The actual transfer of data between virtual memory and the physical storage devices is performed by

31 MARCH 1972

File Management System facilities which may be called from an Apple program or invoked by the system command language.

The virtual memory mechanism provides a large enough storage space that a user can reference all of his data without ever having to do any explicit I/O operations. Since the actual data transmission is transparent to the user, the remaining user-controlled portion of the process is called virtual I/O, which consists of internal memory-to-memory data manipulation and data conversion.

Files may be used for storing many diverse categories of information, ranging from simple collections of card-images to complicated data structures including libraries of executable programs. But regardless of file contents, all files are handled in the same way by the File Management System. Any file must be opened or mapped into virtual memory before any information within the file can be referenced. Saving a file causes a standard File Management procedure to store the file. Closing a file releases the virtual memory space for other usage.

From a programming viewpoint, there are two basic types of files in Apple, sequential files and structured files, each involving an entirely different mode of operation and using separate sets of Apple statements. Sequential files can be considered to be a continuous string of characters such as an output listing or an input deck of punched cards. The programmed processing of sequential files implies an extensive amount of data conversion and use of text-editing facilities normally used for generating reports. Structured files, on the other hand, can be considered as a "carbon-copy" of a portion of the program's virtual memory which can be saved and restored. There is no implied data conversion in the handling of structured files.

Sequential Files

Sequential files have the following properties and constraints:

1. The programmer can not directly reference the data within a sequential file. Input data can only be read via the GET statement, and output data can only be written via the PUT statement.
2. There is no random access to data within a sequential file. The data can only be read or written in a sequential forward direction. Once

31 MARCH 1972

input data has been read, there is no way to back up and re-read the data except to close and re-open the file and start over at the beginning of the file.

3. All data in a sequential file is assumed to be in character form and may contain control characters. (Appendix 4 contains a list of the allowed graphic and control characters.)

Structured Files

Structured files contain an organized collection of data along with the stored hierarchical relationships between data entities. The creation, manipulation, and structural-ordering of data within a structured file are under the direct control of the programmer.

Structured files have the following properties:

1. All data elements within a structured file must be based variables or entities.
2. The data elements can be referenced in a random order. In addition, the FIND and FOR EACH statements provide searching capabilities for entities.
3. Although the associative data handling capabilities are normally used to reference the data within structured files, standard locator qualification of based variables may alternatively be used provided that the proper structure declarations have been made.
4. The ALLOCATE and FREE statements must be used to allocate and free data elements within structured files.
5. The deletion of a data entity having associative relationships includes the automatic removal of those entities that are dependent upon the deleted entity.

File Variables

The Apple file-handling statements refer to a particular file by means of a file variable. The file variable is an identifier which has been declared with the FILE VARIABLE attribute.

31 MARCH 1972

Example:

```
DECLARE INPUT1 FILE VARIABLE,  
        ITEM CHARACTER(20);  
GET FILE(INPUT1) EDIT(ITEM) (A(20));
```

Apple requires that a file must be explicitly opened before any references can be made to the data within the file. There is no implicit opening of a file when the first file-handling statement is executed. The open facility associates a file variable with a particular file. If a file variable has not been associated with a file before a file-handling statement uses the variable, the UNDEFINEDFILE condition is raised.

When the processing of a file is complete, the file should be closed, releasing the facilities established during the opening of the file. A file will be closed automatically on termination of the program that opened it, if it has not been explicitly closed before termination. Since the association between the file variable and the actual file is established dynamically at execution time, a file variable can be associated with different files at different times during the execution of a program.

See the File Management Manual for a complete description of the open, close, and save facilities.

31 MARCH 1972

SEQUENTIAL FILE HANDLING

The format-directed data transmission facilities available for handling sequential files are provided by the GET and PUT statements. These statements provide conversions and data transmission between the internal form of program variables and the character form of the external file data.

Use of GET and PUT Statements

The format-directed mode of data transmission to sequential files uses only one input statement, GET, and one output statement, PUT. The statements may use either of two options to specify the source or target of the data. The FILE option specifies the name of the file variable associated with the opened sequential file on which the operation is to take place. The STRING option specifies the name of the character string variable which is the source or target of the data. A GET statement gets the next series of data items from the source, and a PUT statement puts a specified set of data items into the target. The variables to which input data items are assigned, and the variables or expressions from which output results are transmitted, are specified in a data list with each GET or PUT statement. On input, the format list contains format-items which specify the number of characters to be assigned to each input variable and describe the characteristics of the input data. On output, the format list defines the format that the output data is to have in the target. No type conversion is performed by the GET and PUT statements.

Data Specification

A data specification is used in GET and PUT statements to specify a list of variables and expressions that are to be converted according to a specified format and are to be transmitted to or from a designated file or string variable.

General format:

data-specification ::= EDIT (data-list) (format-list)

General rules:

1. The general rules for data lists and format lists are given later in this chapter under "Data Lists" and "Format Lists".
2. On output, the data items to be transmitted are defined by the data list of the PUT statement. The value of each data item in the data list is converted to the format specified by the associated format-item in the format list.
3. Input data is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by an associated format-item in the format list. The conversion is performed according to the specified format-item. The resulting value is then assigned to the corresponding variable in the data list of the GET statement.
4. For either input or output, the first data format-item is associated with the first item in the data list, the second data format-item with the second item in the data list, and so forth. If the number of data format-items is less than the number of items in the data list, the format list is reused starting at the beginning.
5. The specified transmission is complete when the last item in the data list has been processed using its corresponding format-item. Subsequent format-items, including control format-items, are ignored.

Data Lists

The data specification of a GET or PUT statement requires a data list to specify the variables to which input values are assigned or the variables or expressions from which output results are transmitted.

General format:

data-list ::= element [, element] ...

Syntax rules:

The nature of the elements depends upon whether the data list is used for input by a GET statement or for output by a PUT statement. The rules are as follows:

1. Each data-list element of an input data specification must be a scalar variable.
2. For an output data list, each element must be a scalar expression.
3. The elements of a data list must be of arithmetic or string data type.
4. A data list must always be enclosed in parentheses.

31 MARCH 1972

Format Lists

The data specification of a GET or PUT statement requires a format list to specify the external format for every item in the data list.

General format:

```

format-list ::=
    format-list-item [, format-list-item]...

format-list-item ::= {
    format-item
    n format-item
    n(format-list)
}

```

Syntax rules:

1. The letter n represents an iteration factor, which specifies that the associated format-item is to be used n successive times. The associated format-item is that item or format list of items to the right of the iteration factor. The iteration factor must be an unsigned decimal integer constant.
2. There are two categories of format-items: data format-items and control format-items. Each format-item of a format list must be an allowable type for either category as listed in the table on the next page.

31 MARCH 1972

<u>Type of Format-Item</u>	<u>General Format</u>
Data Format-Items:	
Fixed-point	F (w [,d [,p]])
Floating-point	E (w , d [,s])
Character string	A [(w)]
Bit string	B [(w)]
Control Format-Items:	
Spacing control	X (n)
Column positioning	COLUMN (n)
Line skipping	SKIP [(n)]
Line positioning	LINE (n)
Top of new page	PAGE

The following notation is used in the definition of format-items:

The letter w represents the total number of characters in the field (including a possible sign character, decimal point, blanks, and the letter E denoting an exponent).

The letter d represents the number of fractional digits to the right of a decimal point; it may be omitted for integers.

The letter p specifies a scaling factor, which may be positive or negative, to be used with the F format-item.

The letter s is optionally used in the E format-item to specify the number of significant digits in the coefficient of a floating-point number.

The letter n represents an integer value used by the control format-items to specify number of characters or number of lines.

Each of the quantities w, d, p, s, and n must be specified by a decimal integer constant; w, d, s, and n must be unsigned.

31 MARCH 1972

Data Format-Items

A data format-item describes the character representation of a single data item in the sequential data file or character string specified by a GET or PUT statement. In the format list of a GET statement, each data format-item specifies the number of characters being used to represent an input data item and describes the way those characters are to be interpreted. If the characters in the input string cannot be interpreted in the manner specified, the CONVERSION condition is raised. During execution of a PUT statement, the value of each associated element in the data list is converted to the character representation specified by the data format-item and is inserted into the output character string.

The conversion is defined to occur between the character representation specified by the data format-item and the internal representation of the associated variable or expression. No data type conversion is performed.

Blanks are not automatically inserted into the target to separate the output data items. Arithmetic data is right-adjusted in the format-specified output field. Leading blanks will be inserted in the converted output string if the specified field-width *w* allows for them. If truncation of significant digits due to inadequate field-width specification occurs during the output of arithmetic data items, the ERROR condition will be raised.

String data is left-justified in the specified output field with truncation or padding with blanks occurring on the right.

Fixed-point format-items describe decimal arithmetic data.

General format:

fixed-point-format-item ::= F(w [, d [, p]])

The options referred to in the following rules are:

F(w) : option 1
F(w, d) : option 2
F(w, d, p) : option 3

General rules:

1. On input, the data item in the source is the character representation of an optionally signed decimal arithmetic number anywhere in a field of width w. Leading and trailing blanks are ignored, but if the data consists only of blanks, zero is assumed.

Option 1 is treated like F(w, 0).

In option 2, if no decimal point appears in the number, it is assumed to appear immediately before the last d digits (trailing blanks are ignored). If a decimal point does appear, it overrides the d specification.

In option 3, the scaling factor effectively multiplies the value of the item in the source by 10 raised to the value of p. If p is positive, the number is treated as though the decimal point appeared p places to the right of its given position. If p is negative, the data is treated as though the decimal point appeared p places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it is given, or by d, in the absence of an actual point.

If the number in the source exceeds the allowed range, the ERROR condition is raised.

2. On output, the result is right-adjusted in a field

31 MARCH 1972

of width w . If the value of d is too small to contain all the low-order digits, the result is truncated.

In option 1, only the integer portion of the number is written; no decimal point appears.

In option 2, both the integer and fractional parts of the number are written. If d is greater than 0, a decimal point is inserted before the right-most d digits, and the value is appropriately positioned. Trailing zeros are supplied if the number of fractional digits is less than d (where d must be less than w). If the absolute value is less than 1, a zero precedes the point; if w is not large enough to include the zero, the ERROR condition will be raised.

In option 3, the scaling factor effectively multiplies the internal data value by ten raised to the power of p before it is edited into its external character representation.

For all options, if the value of the data item is less than zero, a minus sign will be prefixed to the character representation in the data stream; if it is greater than or equal to zero, no sign will appear. Therefore, for negative values, w must encompass both sign and decimal point. If the length of the data item is greater than w , the ERROR condition is raised.

3. The variable or expression associated with a fixed-point format-item must be arithmetic.

Floating-point format-items describe the external representation of decimal arithmetic data in floating-point format (coefficient and exponent).

General format:

floating-point-format-item ::= E (w, d [, s])

General rules:

1. When used in a GET statement, the input data item is the character representation of an optionally signed decimal floating-point number located anywhere within the specified field of width w. Leading and trailing blanks are ignored. If the entire field is blank, it is treated as zero.

The external input form of a floating-point number is:

[±] coefficient [E [±] exponent]

The "coefficient" must be an unsigned decimal arithmetic constant. If no decimal point appears in the coefficient field, the decimal point is assumed to be before the rightmost d digits. If a decimal point does appear in the input number, it overrides the decimal point placement specified by d.

The "exponent" must be an unsigned decimal integer constant. If the exponent and prefix letter E are omitted, a zero exponent is assumed.

The width of the input field, expressed by w, includes the character positions for the exponent field, the optional signs, the letter E, a possible decimal point in the coefficient, and any leading or trailing blanks.

2. When used in a PUT statement, the output data is inserted in the specified field after being converted to the following general format:

[-] {s-d digits} . {d digits} E {±} exponent

The exponent is a decimal integer constant that may range from -8630 to +8630. The exponent is automatically adjusted so that the leading digit

31 MARCH 1972

of the coefficient is non-zero. At least one non-fractional digit will always appear; if $\underline{d} = \underline{s}$, a single zero appears to the left of the decimal point. When the value is zero, one zero digit appears to the left of the point and \underline{d} zero digits to the right of the decimal point and the exponent appears as E+0.

The output field-width \underline{w} must be large enough to contain \underline{s} significant digits plus the decimal point, the letter E, an exponent and its sign, and a possible leading minus sign if the data item is negative. Thus, $\underline{w} \geq \underline{s} + 8$ for negative values and $\underline{w} \geq \underline{s} + 7$ for non-negative values of the data item. However, if no fractional digits are specified ($\underline{d} = 0$), the decimal point is not used and the above requirements for field-width are reduced by 1. If any significant digits or the sign is lost because \underline{w} is too small, the ERROR condition is raised.

If the number of significant digits \underline{s} is omitted, it is assumed to be $\underline{d} + 1$.

3. The number of significant digits \underline{s} specified in the E format-item E(w,d,s) must be less than 16 digits. If \underline{s} is omitted and the E(w,d) form is used, \underline{d} must be less than 15.
4. The variable or expression associated with a floating-point format-item must be arithmetic.

The bit-string format-item describes the character representation of a bit string.

General format:

bit-string-format-item ::= B [(w)]

General rules:

1. The field-width, w, is an integer constant which specifies the number of character positions in the data stream that contain (or will contain) the character representation of a bit string. Each bit is represented by the character 0 or 1. The field-width is always required on input; if w = 0, a null string is assumed. If the w option is omitted on output, w is taken to be the length of the associated bit string; the resulting data item completely fills the output field.
2. When executing a GET statement, the input data item may occur anywhere within the specified field of width w. Leading and trailing blanks are ignored. If the entire field is blank, the CONVERSION condition is raised. If w differs from the declared length of the variable, the input is extended with zeros or truncated on the right. Any character other than 0 or 1 in the input data string, including embedded blanks, enclosing quotation marks, or the letter B, will raise the CONVERSION condition.
3. During execution of a PUT statement, the value of the associated expression from the output data list is left-justified in the specified output field. If necessary, truncation or extension with blanks is performed on the right. No enclosing quotation marks are inserted, nor is the identifying letter B.
4. The variable or expression associated with a bit-string format-item must be a bit string.

31 MARCH 1972

The character-string-format-item describes the external representation of a string of characters.

General format:

character-string-format-item ::= A [(w)]

General rules:

1. The character string is contained in w character positions of the external data stream. The field-width w is always required on input, but it is optional for output.
2. During an input operation, w characters are obtained from the input data stream. If w = 0, a null string is assumed. If quotation marks appear in the input data, they are treated as ordinary characters in the string. The input characters are extended with blanks or truncated on the right.
3. During the execution of a PUT statement, the associated value from the output data list is truncated or extended with blanks on the right to the specified field-width w before being placed into the output field. Enclosing quotation marks are not inserted. If the field-width w is not specified, it is assumed to be equal to the character-string length of the associated data item.
4. The variable or expression associated with a character-string format-item must be a character string.

31 MARCH 1972

2. The COLUMN format-item specifies that the file is to be positioned to the nth column of the current line. If the file is an output file, blank characters are inserted into the stream until the nth column of the line is reached. If the file is an input file, characters are ignored until the nth column is reached. If the file is already positioned beyond the nth column of the current line, the file is positioned to the nth column of the next line. If n is less than 1 or greater than the line size of the file, n is assumed to be 1.

A printing-format-item specifies that the next data item transmitted is to appear on a new page or on a particular line on a page.

General format:

$$\text{printing-format-item} ::= \left\{ \begin{array}{l} \text{LINE (n)} \\ \text{PAGE} \end{array} \right\}$$

General rules:

1. Printing-format-items may only be used with sequential output files.
2. The LINE format-item specifies that the next data item is to be printed on the nth line on a page of an output file. Blank lines will be inserted so that the next line will be the nth line of the current page (or the next page, if the current line number is greater than or equal to n).
3. The PAGE format-item specifies that a new page is to be established in the print output file. The establishment of a new page implies that the next printing is to be on line one.

Note that X and COLUMN specify, respectively, relative horizontal spacing and absolute horizontal spacing. Similarly, SKIP and LINE specify relative vertical positioning and absolute vertical positioning.

STRUCTURED FILE HANDLING

A structured file is one that is made up of data elements that are related by defined associations. The elements themselves may be referenced in random order (as compared to sequential) and a formal organization is preserved in order to represent the relationships between elements.

All data elements within a structured file must be ENTITY or BASED variables. That is, storage space within the file is created through use of the ALLOCATE statement and deleted through use of the FREE statement. Locator qualification must be used to refer to a particular generation of ENTITY or BASED variable within a file. If the variable is to be used to form an association (i.e. relationship) between data elements, it is referred to as an ENTITY. The allocation of a variable does not imply the establishment of associations. The INSERT and REMOVE statements must be used to establish associations between groups of entities (called SETS) based upon some common relationship. The FIND and FOR EACH statements may be used to search a set for an entity with particular characteristics.

Storage Management

The allocation and freeing of storage space within a structured file will be controlled by the ALLOCATE and FREE statements. A file is activated and saved on external storage devices in multiples of 4096 bytes (1 page). The ALLOCATE statement will always try to assign the request for space to the lowest address that will accommodate the request. If all current active pages of the structured file will not accommodate the request, a new page will automatically be activated up to a user defined limit. When storage space is released via the FREE statement, adjacent unused space will be combined, and unused pages will automatically be deactivated. A deactivated page will not be saved on permanent storage. In general, recovery of unused storage space within a structured file will not be required because of the dynamic activation and deactivation of pages.

By default, the allocation and freeing of BASED variables will be done in a scratch file. Optionally, the programmer may specify a particular file through use of the IN clause in both the ALLOCATE and FREE statements. In addition, the

31 MARCH 1972

programmer may exert some control over the location of the allocation. The ALLOCATE statement provides an optional means for specifying that a BASED variable shall be NEAR to or REMOTE from another based variable or AT a particular location. NEAR implies that the allocation shall be made in the same page as the referenced variable. If this cannot be done, the normal space allocation rules will apply. The REMOTE option implies that the allocation shall be made in a new page.

Examples:

```

DECLARE F1 FILE VARIABLE,
      (A, B, C) POINTER,
      S(5) FIXED BASED(A),
      T(8) FLOAT BASED(A),
      ...
ALLOCATE S IN (F1);
...
ALLOCATE T IN (F1) NEAR (A);
...
ALLOCATE T SET (C);
...
FREE T IN (F1);
...
FREE C -> T;

```

Entities

A BASED structure variable that can be used in the establishment of relations or associations is known as an ENTITY variable. An entity identifier is a maximum of 8 alphanumeric characters or break symbols, and must begin with an alphabetic character. An example of an entity declaration is:

```
DECLARE POINT ENTITY (P);
```

A system function will be provided to store the structure declaration for the POINT variable in a file. At compile time this declaration will replace the entity declaration. Thus, the entity POINT might be replaced by the declaration:

```

DECLARE 1 POINT BASED(P),
        2 H OFFSET(F),
        2 M OFFSET(F),
        2 X FLOAT,
        2 Y FLOAT,
        2 Z FLOAT;

```

Presumably, standard declarations will be established for projects, thus assuring a continuity of data item declarations across procedures. The variables F and P must have been declared by the programmer.

Since an entity is a pre-defined type of structured based variable, normal pointer qualification may be used to refer to different generations of an entity.

Example:

```

DECLARE (P1,P2) POINTER,
        POINT ENTITY (P1);

IF POINT.X < 8.5 THEN
    P2 -> POINT.Y = 5;

```

Sets

By definition, a set is an ordered collection of entities. Each entity is referred to as a member of the set. Sets have a unique identification that may be expressed in terms of the file or entity that contains the set and the name of the set. The set name is a maximum of 8 alphanumeric characters or break symbols and must begin with an alphabetic character. An example of a set declaration is:

```
DECLARE BNDRY SET;
```

Generations of the named set may be referenced as follows:

```

locator-variable
locator-variable -> set-name
locator-variable -> (character-string-expression)

```

If only a locator-variable is used to reference a set, the locator must have been assigned a value through execution of a LET statement. If a locator-variable is used in conjunction with a set-name or character-string-expression whose

31 MARCH 1972

value is a set-name, the locator-variable must reference an entity that contains the named set.

A set that is contained by the file in which it resides is referred to as a FILE_SET. An example of a FILE_SET declaration is:

```
DECLARE TERM FILE_SET;
```

and may be referenced as follows:

```
locator-variable
set-name [OF (file-variable) ]
character-string-expression [OF(file-variable) ]
```

If only a locator-variable is used to reference the set, the locator must have been assigned a value through execution of a LET statement. Alternatively, a FILE_SET may be referenced by its name or by a character-string-expression. If the optional OF (file-variable) clause is present, the FILE_SET in the specified file will be referenced. If no file is specified, a "current" file will be interrogated for the referenced set. The "current" file may be specified by the programmer by setting the STATIC EXTERNAL variable SYSFILE.CURRENT equal to a particular file-variable.

```
DECLARE 1 SYSFILE STATIC EXTERNAL
        2 SCRATCH FILE VARIABLE,
        2 CURRENT FILE VARIABLE,
        2 STATIC FILE VARIABLE;
```

An entity that is an element of a set is said to be a member of the set. An entity may be the member of one or more sets having the same name or different names. Since sets are ordered, the programmer can reference the first, second, or last member (entity) of a set. Also, since a set may or may not be homogenous, it makes sense to talk about the first or second member of a set of a particular type. Entities may also contain one or more sets, provided each set has a different name. In the case of a named set owned by an entity-variable, many generations of the set can exist in the same file. However, only one generation of a named FILE_SET may exist in a file.

A set cannot be allocated or freed. When the INSERT statement is first used to make an entity a member of a set, the set will automatically be created. Similarly, if all members of a set are removed, the set will be deleted.

Creating and Deleting Associations

The INSERT statement is used to include a generation of an entity as a member of a particular set. Any entity may be inserted as a member of any set. Hence, a set may contain many generations of the same entity or many different entities. An entity can also be inserted onto more than one set. The first entity inserted into a set will be the first member.

Because sets are ordered, it also is possible to specify a position that an entity is to assume as a member of a set. The BEFORE, AFTER, FIRST, and LAST clauses of the INSERT statement allow the programmer to specify a relative position at which to make the insertion. If the members of a set are numbered (1,2,3) and a new entity is inserted after entity 2, it will become member 3 and the previous member 3 will become number 4.

Example:

```

DECLARE (P1,P2) POINTER,
        POINT ENTITY (P1),
        CURVE FILE-SET;

        ...
INSERT POINT ON CURVE;
        ...
INSERT P2 -> POINT ON CURVE;

```

The REMOVE statement is used to remove a member entity from one or more sets. If the ALL option is used, the member entity will be removed from all sets of which it is a member. If an entity is freed, it will automatically be removed from all sets of which it is a member. If a deleted entity contains one or more sets, all members of these sets will be deleted provided they are members of no other sets. This recursive process is continued until no more entities can be deleted. Member entities will be deleted regardless of their membership in other sets if the INCLUSIVE option is applied to the FREE statement.

31 MARCH 1972

Example:

```
DECLARE (A, B) POINTER,  
        ALL BUILTIN,  
        LINE ENTITY (B),  
        SETNAM CHAR(8);  
...  
REMOVE LINE FROM ALL;  
...  
SETNAM = 'LINE_SET';  
REMOVE A -> LINE FROM SETNAM;
```

Searching a Set

Sets may be searched using either the FIND or FOR EACH statements. The FIND statement is used to locate a particular member of a set or entity that contains a set. If the search is successful, the FIND statement will set a locator-variable to reference a generation of an entity-variable. Since sets are ordered, the FIND statement provides a means for specification of the *n*-th member. The user may optionally search for an entity of a particular type. The absence of the CONTAINING option assumes the search will be made for a member entity. If the CONTAINING option is used, the search will be made for an entity that contains the referenced set. Searches that are unsuccessful will either raise the FIND condition or cause control to pass to an optional ELSE statement.

Conditions may be placed on the extent of the search performed by the FIND statement. The optional WITH clause allows a relational-expression to be evaluated for every entity included in the search of a set. The relational-expression may involve attributes of the current entity. If the relation is true, the entity will be counted in the search. If the relation is false, the entity will not be counted in the search and the search will continue. If the optional UNTIL clause is used, the search will be terminated if the UNTIL relational-expression is true.

31 MARCH 1972

Example:

```
DECLARE (P1,P2) POINTER,  
        DATA  ENTITY (P1);  
        ...  
FIND DATA = (I) ENTITY IN 'SETA';  
        ...  
FIND P2 = (1) ENTITY CONTAINING P1 ON P2 -> S1;
```

If a group of statements is to be executed for all or part of the members or containers of a set, the FOR EACH statement may be used to delimit the start of the group. The forms of the FOR EACH statement have a one to one correspondence with the forms of the FIND statement. In practice, the function of a FOR EACH statement may be replaced by the corresponding FIND statement within a DO WHILE group. In all cases, the failure to find the next member or container of the set is sufficient cause to terminate the group.

Examples:

```
DECLARE (A,B) POINTER,  
        ORDER ENTITY (A);  
        ...  
FOR EACH B -> ORDER=ENTITY ON TSET;  
    I = I + 1;  
    N(I) = B -> ORDER.NAME;  
END;
```

Associative Data Built-In Functions

The Apple language includes a collection of 8 built-in functions to aid in the manipulation of entities and sets. The operation of each function is described in Appendix 1 of this manual.

CHAPTER 7 - INTERRUPT HANDLINGINTRODUCTION

During the course of program execution, the program may be interrupted by the occurrence of an error or an action which is generated from an external source. There are two types of occurrences which can cause an interrupt:

1. the raising of a condition, and
2. the completion of an event.

The circumstances which may cause a condition to be raised are related to instruction execution; thus the program knows "when" and "where" to expect a potential condition to be raised. An event is associated with one or more external actions that can occur on a peripheral device. Therefore, the Apple program does not know "when" or "where" to expect an event to become complete.

CONDITIONS

There are two types of conditions: system conditions and programmer-defined conditions. Conditions may be specified in the ON, REVERT, and SIGNAL statements. The ON and REVERT statements allow the programmer control over the action to be taken when a condition is raised. A complete list of the conditions, the circumstances under which they may be raised, and the standard system action taken in the absence of programmer-specified action, appears in Appendix 2.

System Conditions

Each system condition is identified with a unique identifier suggestive of that condition (e.g., ZERODIVIDE specifies the condition raised whenever an attempt is made to divide by zero). This collection of identifiers is an intrinsic part of the Apple language, but the identifiers are not reserved.

31 MARCH 1972

These identifiers are keywords when used in the ON, REVERT, and SIGNAL statements.

When a system condition is raised, and no programmer-specified action exists, the standard system action for that condition is taken. The ON statement can be used to specify some other action, the REVERT statement can be used to delimit the scope of an ON statement, and the SIGNAL statement can be used to simulate the raising of a system condition. The use of these statements appears later in this chapter (also see Chapter 8 - Statements).

Programmer-Defined Conditions

Programmer-defined conditions may be used in testing and debugging programmer-specified action. A programmer-defined condition is declared with the CONDITION attribute. The execution of a SIGNAL CONDITION statement is the only way to raise a programmer-defined condition.

EVENTS

An external action may be referenced in a procedure through use of an event. The method of associating an event with one or more external actions is defined in the Reactive Terminal User's Manual. An example of an external action is a user pushing a function key on a graphic terminal, the selection of a graphic entity with the light pen, etc. Events may be specified in the ON, REVERT, SIGNAL, and WAIT statements. The LOCK and UNLOCK statements are used to protect portions of program execution from event interrupts.

Event Declarations

An event identifier may be declared with any scope. The declared event may then be associated with one or more external actions.

31 MARCH 1972

Example:

```

A: PROC;
  DECLARE EV1 EVENT,
         EV2 EVENT EXTERNAL;
  ...
  a CALL associating EV1 with external action C1
  a CALL associating EV2 with external action C2
  ...
  CALL B;
  ...
END A;

B: PROC;
  DECLARE EV2 EVENT EXTERNAL;
  ...
  a CALL associating EV2 with external action C3
  ...
END B;

```

Event EV1, by the rules of default, has internal scope. Thus, EV1 is known only within procedure A. Event EV2 has been declared to have external scope. Thus, EV2 is common to both procedures, A and B. Note that the same event may be associated with different actions by different procedures.

Event States

Every event has two states associated with it: the completion state and the delay state. Thus, an event can be complete or incomplete and delayed or nondelayed. Upon declaration an event is initialized to be incomplete, and delayed. Each state has a value of '1'B or '0'B.

Completion State

An event becomes complete when an external action associated with the event occurs. Once an event becomes complete it remains complete until the program references the event in the ONPTR built-in function which is described later in this chapter. The COMPLETION built-in function can be used to test whether an event is complete or incomplete (see Appendix 1). The SIGNAL statement can be used to set an event complete. There is no COMPLETION pseudo-variable.

Delay State

The delay state of an event determines when a program is ready to react after an associated external action becomes complete. If the event is delayed, the program reacts to event completion synchronously; if the event is nondelayed, the program reacts asynchronously. An event is initialized to be delayed. When a delayed event becomes complete, the completion is enqueued so that the program may react to this completion at some future time. When a nondelayed event becomes complete, the on-unit associated with the event is executed. If no on-unit is found, the ERROR condition is raised.

The delay state can be changed by using the DELAY pseudo-variable (see Appendix 1). This is the only way to change the delay state of an event.

Example:

```
DELAY (EV1) = '0'B;
```

The above example will set the event EV1 to nondelayed. The DELAY built-in function can be used to test whether an event is delayed or nondelayed.

Use of the ONPTR Built-in Function

When an event becomes complete, a block of information (Event Completion Block) about that completion is saved (see the Reactive Terminal User's Manual for details about this information). The ONPTR built-in function provides a means of accessing the saved information (see Appendix 1 - Built-in Functions). The value returned by the ONPTR built-in function specifying an event is a pointer to the Event Completion Block saved when that event became complete. The reference to the event in the ONPTR built-in function also resets the event to incomplete. Since event occurrences may be queued, another ONPTR reference to the same event may return a pointer to another information block. If another occurrence has not been queued, a null pointer is returned.

31 MARCH 1972

USE OF INTERRUPT-HANDLING STATEMENTS

The interrupt-handling statements are the ON, REVERT, SIGNAL, WAIT, LOCK, and UNLOCK statements. The ON, SIGNAL, and REVERT statements are used with both conditions and events, while the WAIT, LOCK, and UNLOCK statements are used only with events.

Use of the ON Statement

A system action exists for every condition or event. The ON statement is used to specify alternative action that is to be taken when a specified condition is raised or event becomes complete.

When an ON statement that is internal to a given block is executed, the specified action remains in effect until overridden or until termination of the block containing the ON statement. An established action passes from the defining block to all dynamically descendent blocks, and the action remains in force until overridden by execution of another ON statement for the same condition or event. If overridden, the new action remains in force only until that block is terminated. When control returns to the activating block, all established actions that existed at that point are re-established. This makes it impossible for a subroutine to alter the interrupt action for a block that invoked the subroutine.

If more than one ON statement for the same condition or event appears in the same block, each execution of an ON statement overrides the action established by previous execution of other ON statements. No re-establishment is possible, except through execution of another ON statement with an identical action specification.

Control passes to the on-unit in an ON statement only when the specified condition is raised or the specified event becomes complete. Any variables which appear in the on-unit have the attributes and the environment of the block dynamically encompassing the ON statement unless they are declared in the on-unit. If the on-unit is a null statement, no action is taken when the condition or event occurs. In some situations, the programmer may want to specify his

31 MARCH 1972

own action for a given condition or event, to have it hold for part of the execution of the program, and then to have this specification nullified and allow the standard system action. In this case he may use the keyword SYSTEM as the action specification.

Example:

```
X: PROCEDURE;
  DECLARE (A, B) FIXED;
  ...
  ON OVERFLOW
    BEGIN;
      PUT FILE(F1) EDIT(A, B) (F(2), F(6));
    END;
  ...
  Y: BEGIN;
    DECLARE (A, B) FLOAT;
    ...
    END Y;
  ...
END X;
```

This example illustrates the effect of establishment of the generation of variables at the time an ON statement is executed. If the OVERFLOW condition should arise, the values transmitted by the PUT statement in the on-unit will be the values of the variables A and B that are declared in the outer block. This is true even if the OVERFLOW condition should arise during the execution of the begin block Y, where A and B have been redeclared.

Example:

```
A: PROCEDURE;
  ...
  ON OVERFLOW
    BEGIN;
      DECLARE NUMBOV FIXED STATIC INITIAL (0);
      NUMBOV = NUMBOV + 1;
      IF NUMBOV = 100 THEN GO TO OVERR;
    END;
  ...
  ON OVERFLOW;
  ...
OVERR: ON OVERFLOW SYSTEM;
  ...
END A;
```

31 MARCH 1972

In the above example, assume that the program consists only of procedure A; that the three ON statements are the only ON statements involving the OVERFLOW condition, that they are internal to procedure A, and that they are executed in their physical order. When program execution begins, the OVERFLOW condition is enabled by the system; any floating-point overflow condition that occurs before the first ON OVERFLOW statement is executed will result in an interrupt, with standard system action. However, the execution of the first ON OVERFLOW statement establishes the action specified in the SIGNAL block. (The number of overflows is counted and if this number has not reached 100, the action is finished.) Any OVERFLOW interrupts will receive this action until the second ON OVERFLOW statement is executed. The action specified here is a null statement; any subsequent OVERFLOW interrupts will effectively be ignored until control reaches the third ON OVERFLOW statement, which re-establishes the standard system action.

The ON statement may specify a programmer-defined condition. Control passes to the on-unit only when a SIGNAL statement specifying the same condition is executed.

Example:

```

A: PROCEDURE;
  DECLARE ABC CONDITION;
  ***
  ON CONDITION(ABC)
    GO TO L;
  ***
  SIGNAL CONDITION (ABC);
  ***
END A;

```

The execution of the ON statement establishes the action to be taken when the SIGNAL statement is later executed.

The ON EVENT statement is used to establish an asynchronous action which takes place when an event becomes complete. The event specified in the ON statement must be nondelayed for control to pass to the on-unit when an interrupt occurs. If the event is delayed, execution will continue when the interrupt occurs; control will not pass to any on-unit.

31 MARCH 1972

Example:

```

A: PROCEDURE;
  DECLARE XYZ EVENT;
  ...
  a CALL associating function key 3 with XYZ
  ...
  ON EVENT (XYZ)
    BEGIN;
    ...
    END;
  ...
  DELAY (XYZ) = '0'B;
  ...
END A;

```

The event XYZ is initialized to be delayed. If the function key is depressed before the statement associating XYZ with function key 3 is executed, the interrupt is ignored. If the interrupt occurs before the DELAY (XYZ) = '0'B; statement is executed, execution will continue normally; control will not pass to the on-unit. If the interrupt occurs after the DELAY (XYZ) = '0'B statement is executed, control will pass to the on-unit of the ON statement. The execution of a DELAY (XYZ) = '1'B; statement will re-establish synchronous action when the interrupt occurs. The delay value at the time the external action occurs determines whether control passes to the on-unit.

Use of the REVERT Statement

The REVERT statement may be used, following an ON statement, to reinstate an action specification that existed in the nearest dynamically encompassing block at the time the descendant block was invoked. The REVERT statement does not re-establish the completion or delay values of an event, only the specified on-unit. Since there may only be one active on-unit in a block for the same interrupt, the REVERT statement cannot revert back to a previously active on-unit in the same block, only to the active on-unit in the nearest dynamically encompassing block.

31 MARCH 1972

Example:

```
A: PROCEDURE;  
    ON ZERODIVIDE  
        GO TO AERR;  
    ...  
    CALL B;  
    ...  
END A;  
B: PROCEDURE;  
    ON ZERODIVIDE  
        GO TO BERR;  
    ...  
    REVERT ZERODIVIDE;  
    ...  
END B;
```

In the above example, if a ZERODIVIDE condition occurs in procedure B after execution of the ON statement, an interrupt will take place with the resulting action GO TO BERR. After execution of the REVERT statement, the action as specified by the ON statement in procedure A is reinstated. Program control remains in procedure B, but any subsequent ZERODIVIDE condition that occurs in procedure B will cause an interrupt with the action GO TO AERR and result in the termination of block B.

Use of the SIGNAL Statement

The SIGNAL statement simulates the occurrence of the specified condition or event. It can be used to test and debug the action specification of an ON statement. The SIGNAL statement is the only way to pass control to the on-unit of an ON statement specifying a programmer-defined condition.

Use of the WAIT Statement

The WAIT statement is used to relinquish control and to synchronize the processing of delayed event completions. The WAIT statement cannot be used with conditions or nondelayed events.

31 MARCH 1972

Example:

```

A: PROCEDURE;
  DECLARE EC_1 EVENT,
          EC_2 EVENT;
  ...
  a CALL associating function key 1 with EC_1
  a CALL associating function key 1 with EC_2
  ...
  WAIT (ANY);
  ...
END A;

```

In the above example, the program will go into a wait state until one of EC_1 and EC_2 becomes complete, at which time processing continues.

Use of the LOCK and UNLOCK Statements

The LOCK and UNLOCK statements are used to place a program or part of program execution into locked status. When a program is in locked status, all asynchronous events will be queued. No on-units specified in ON EVENT statements will be invoked during locked status. On-units for conditions are not affected by the LOCK and UNLOCK statements.

Example:

```

A: PROCEDURE;
  DECLARE X1 EVENT;
  a CALL associating external action with X1
  ...
  ON EVENT(X1) ABC = ABC + 1;
  DELAY(X1) = '0'B;
  LOCK;
  ...
  UNLOCK;
  ...
END A;

```

In the above example, if the event X1 becomes complete between the LOCK and UNLOCK statement, that completion will be queued until the UNLOCK statement has been executed, at which time the on-unit; ABC = ABC + 1, will be executed.

31 MARCH 1972

CHAPTER 8: STATEMENTSINTRODUCTION

This chapter includes a description of each statement in the Apple language. These descriptions are presented in alphabetic order.

CLASSIFICATION OF STATEMENTS

Statements may be classified into the following logical groups according to the function that they perform: assignment, control, file-handling, declaration, interrupt handling, program structure, storage allocation, and the null statement.

Assignment statement

The assignment statement is used to evaluate expressions and to assign values to scalars, arrays, and structures.

Control statements

The control statements affect the normal sequential flow of control through a program. The control statements are CALL, DO, END, EXIT, FOR EACH, GO TO, IF, PROCEDURE, RETURN, SIGNAL, and WAIT.

File-handling statements

The GET and PUT statements cause values to be transmitted between sequential files or character strings and specified variables in the program. Associative data structures are built from entities and sets contained in structured data files. The FIND, FOR EACH, INSERT, REMOVE, and LET statements are used to reference and manipulate associative data structures.

Declaration statement

The declaration statement, DECLARE, specifies the attributes to be associated with identifiers.

Interrupt handling statements

There are two kinds of interrupts; internal interrupts or

31 MARCH 1972

conditions, and external interrupts or events. The ON, REVERT, and SIGNAL statements are used with both kinds of interrupts while the LOCK, UNLOCK, and WAIT statements apply only to the external interrupts.

Program structure statements

The program structure statements are: PROCEDURE, BEGIN, END, DO, FOR EACH, and ENTRY. The first three statements delimit the scope of declarations within a program. The DO and FOR EACH statements delimit groups for the purposes of control or repetitive execution. The ENTRY statement provides a secondary entry point for a procedure.

Storage allocation statements

The storage allocation statements are ALLOCATE, CREATE, DELETE, and FREE. These statements obtain and release storage for based variables.

The ALLOCATE Statement

Function:

The ALLOCATE statement causes storage to be allocated for specified based variables.

General format:

allocate-statement ::=

ALLOCATE allocation [, allocation] ... ;

allocation ::=

identifier [IN (scalar-file-variable)]
 [ALIGN (integer-expression)]
 [SET (scalar-locator-variable)]

$$\left[\begin{array}{l} \{ \text{NEAR} \\ \text{REMOTE} \} \\ \text{AT} \end{array} \right] \quad (\text{scalar-locator-variable})$$

General rules:

31 MARCH 1972

1. The "identifier" must be the name of a level-1 scalar, array, or major structure variable with the storage-class attribute BASED.
2. The amount of storage to be allocated is determined by evaluating all bounds of arrays and lengths of strings. Although the extents and initial values of the variable are evaluated at the time of execution of the ALLOCATE statement, the names in these expressions are interpreted in the environment of the DECLARE statement. These expressions may not contain references to the variable being allocated except in the REFER option.
3. The allocation of a based variable has no effect on other generations of the variable. A given generation of a based variable may be accessed by a suitable based reference regardless of allocations of the same based variable made subsequently. The allocation of a based variable proceeds as follows:
 - a. Bounds and string lengths of all the fields are evaluated in an unspecified order. Expressions preceding the keyword REFER are used as the values of the bounds and string lengths specified by the REFER options.
 - b. Sufficient storage for a generation of the based variable with these evaluated bounds and string lengths is allocated. Should there be insufficient space for the allocation in the file, the STORAGE condition will be raised.
 - c. Within the newly allocated generation, those variables that are objects of REFER options are initialized to the values evaluated in the REFER options. This initialization is performed in an undefined order.
 - d. The locator variable specified in the SET option or, in its absence, the locator variable specified in the BASED attribute of the based variable declaration is assigned a pointer value that identifies the generation just allocated.

31 MARCH 1972

- e. Initial values specified in the declaration of the based variable are assigned to the new generation.
4. The allocation of a based variable involves the based variable to be allocated, a locator variable to identify the new generation, and a file variable if the generation is to be allocated in other than scratch storage. If no SET option is specified, a SET option is assumed to specify the locator variable given in the BASED attribute of the based variable declaration. It is an error if the BASED attribute does not specify a locator variable. If the SET option specifies an offset variable and there is not an IN option, then an IN option that specifies the file variable given in the declaration of the OFFSET attributes is assumed. If no file variable is specified, the program is in error.
5. If the SET option specifies an offset variable, the pointer value identifying the new generation is assigned to the offset variable. The IN option, either in the statement or assumed, must refer to the same file as that specified in the OFFSET attribute of the offset variable declaration.
6. If no IN option is present and none is assumed, the new generation is allocated in a scratch file. In the case of entity variables, the default file is determined from the value of SYSFILE.CURRENT.

```
DECLARE 1 SYSFILE STATIC EXTERNAL,  
        2 SCRATCH FILE VARIABLE,  
        2 CURRENT FILE VARIABLE,  
        2 STATIC FILE VARIABLE;
```

7. If an IN option is present, or is assumed, an attempt is made to allocate the new generation in the designated structured file. If insufficient storage exists, the STORAGE condition is raised.
8. If the NEAR option is present, an attempt will be made to allocate the based variable in the same page as referenced by the locator variable. If the attempt fails, the NEAR option will be ignored.

31 MARCH 1972

9. If the REMOTE option is present, an attempt will be made to allocate the based variable in a new page. If the attempt fails, the REMOTE option will be ignored.
10. If the AT option is present, an attempt will be made to allocate the based variable at the address specified by the locator variable. If the attempt fails, the STORAGE condition is raised.
11. If the ALIGN option is present, the integer-expression will be evaluated to give a value y. An attempt will be made to allocate the based variable at a location whose address is an integer multiple of $\text{FLOOR}(y/64)*64$. If the attempt fails, the STORAGE condition will be raised.
12. On normal return from a STORAGE on-unit, all the options are re-evaluated and the allocation is attempted again.

Examples:

```

DECLARE A(N, M) BASED(P),
        (P,Q) POINTER;
...
N, M = 100;
ALLOCATE A;
...
N = 50;
ALLOCATE A SET(Q);
...

```

This example creates two generations of A, the first is 100 x 100 and the second 50 x 100.

```

DECLARE NAME CHARACTER(200) BASED,
        (P,Q) POINTER;
...
ALLOCATE NAME SET(P);
ALLOCATE NAME SET(Q);
...
P -> NAME = 'ABC';
Q -> NAME = 'XYZ';

```

In this example, two generations of NAME are created, each having a length of 200 characters. The pointer P identifies the first generation and the pointer Q identifies the second.

31 MARCH 1972

```

DECLARE NAME CHARACTER(N) BASED,
        (P,Q) POINTER;
...
N = 100;
ALLOCATE NAME SET(P);
...
N = 200;
ALLOCATE NAME SET(Q);

```

This example differs from the previous one in that the length of NAME is specified by the expression N, thus allowing the length of each generation to be unique. However, because the extents of based variables are evaluated at each reference, the programmer must ensure that N has the proper value when each generation of NAME is referenced.

```

N = 100;
P -> NAME = 'ABC';
N = 200;
Q -> NAME = 'XYZ';
/* P -> NAME = Q -> NAME; */

```

The assignment shown as a comment is illegal because N cannot have the value 100 and 200 at the same time. To relieve the programmer from the burden of maintaining the proper extents when referencing based variables, the REFER option can be used.

```

DECLARE 1 S BASED,
        2 N FIXED BINARY(23),
        2 NAME CHARACTER(M REFER(S.N)),
        (P,Q) POINTER,
        M FIXED BINARY(23);
...
M = 3;
ALLOCATE S SET(P);
M = 4;
ALLOCATE S SET(Q);
P -> NAME = 'ABC';
Q -> NAME = P -> NAME !! 'D';

```

Each allocation causes the length expression M to be evaluated and its value used to create storage for the generation of S being allocated. The value of M is then assigned to the newly allocated generation of S.N. Subsequent references to NAME always use the generation of S.N identified by the pointer used to reference NAME.

31 MARCH 1972

Q -> NAME uses Q -> S.N
 P -> NAME uses P -> S.N

The value of Q -> NAME after the last statement is 'ABCD'.

The Assignment Statement

Function:

The assignment statement is used to evaluate an expression and to assign its value to one or more target variables. The target variables may be scalars, arrays, or structures. The target variables may also be indicated by pseudo-variables.

General format:

$$\text{assignment-statement} ::= \left\{ \begin{array}{l} \text{scalar-assignment} \\ \text{array-assignment} \\ \text{structure-assignment} \end{array} \right\} ;$$

$$\text{scalar-assignment} ::= \left\{ \begin{array}{l} \text{scalar-variable} [, \text{scalar-variable}] \\ \text{pseudo-variable} [, \text{pseudo-variable}] \dots \end{array} \right\} \\ = \text{scalar-expression};$$

$$\text{array-assignment} ::= \text{array-variable} [, \text{array-variable}] \dots \\ = \left\{ \begin{array}{l} \text{array-expression} \\ \text{scalar-expression} \end{array} \right\}$$

$$\text{structure-assignment} ::= \text{structure-variable} \\ [, \text{structure-variable}] \dots \\ = \text{structure-variable} ;$$

Syntax rule:

In the scalar-assignment, the target variables must be scalars. In the array-assignment, the target variables must be arrays. Assignment of structures can only be made between structures that have the same number of elements, n,

31 MARCH 1972

such that for $1 \leq i \leq n$, the i th element of each structure has identical data and aggregate attributes.

General rules:

1. A scalar assignment consists of the following operations carried out in undefined order:
 - a. Subscripts and qualification of the targets are evaluated.
 - b. The expression on the right-hand side is evaluated.
 - c. For each target variable the value of the expression is converted to the characteristics of the target variable according to the rules stated in "Expressions" in Chapter 4 -- Data Manipulation. The converted value is then assigned to the target variable.
2. The following rules apply to string scalar assignment:
 - a. If the target is a fixed length string, the expression value is truncated on the right if it is too long or padded on the right (with blanks for character strings, zeros for bit strings) if the value is too short. The resulting value is assigned to the target.
 - b. If the target is a string with the VARYING attribute and the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right. The target string acquires a current length equal to its maximum length.
 - c. If the target is a character string with the VARYING attribute and the value of the expression is not greater than the maximum length declared for the variable, the value is assigned and the current length of the target string becomes equal to length of the value.
 - d. If the target is the SUBSTR pseudo-variable of a character string with the VARYING attribute, the length of the target character string will not be changed by the assignment.

31 MARCH 1972

- e. If either the source or target string is qualified by a descriptor variable, the length field of the descriptor will override the declared length of the string.
3. The following rules apply to assignments other than string:
- a. If the target is a locator variable, the expression must yield a locator value.
 - b. If the target is a label variable, the expression must be a label constant or label variable. In both cases, the environment of the label will be included in the assignment. The environment of a label constant is created by the activation of the block in which the constant appears. References to a label constant contained in an inactive block produces undefined results.
 - c. If the target is a file variable, the expression can only be a file variable or a function that returns a file variable value.
 - d. If the target is an entry variable, the expression must be an entry constant or entry variable. The environment of the entry will be included in the assignment. The environment of an entry constant is created by the activation of the block in which the constant appears. References to an entry constant (with internal scope) contained in an inactive block produces undefined results.
4. The following rules apply to array assignment:
- a. All target variables must have the same number of dimensions and identical constant bounds. If the expression is an array variable, it must have the same number of dimensions as the target variables and the bounds must be identical.
 - b. If the expression is a scalar-expression, it is evaluated and the value is assigned to all elements of the target variables.

31 MARCH 1972

- c. If either the source or target array is qualified by a descriptor variable, the length field of the descriptor will override the declared length of the array.
5. The following rules apply to structure assignment:
- a. Target and source variables must be left to right equivalent.
 - b. The bounds and lengths of all contained elements must be constant and match at the time of the assignment.
 - c. If either the source or target structure is qualified by a descriptor variable, the length field of the descriptor will override the declared length of the structure.

Examples:

The following example illustrates array assignment:

Given the arrays $A = \begin{bmatrix} 2 & 4 \\ 3 & 6 \\ 1 & 7 \\ 4 & 8 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 5 \\ 7 & 8 \\ 3 & 4 \\ 6 & 3 \end{bmatrix}$

The value of A after the execution of the assignment statement:

$$A = (A + B)**2 - A(1,1);$$

is $\begin{bmatrix} 7 & 79 \\ 98 & 194 \\ 14 & 119 \\ 98 & 119 \end{bmatrix}$

31 MARCH 1972

The following example illustrates string assignment:

```

DECLARE A CHARACTER(5) INITIAL('XZ/BQ'),
        B CHARACTER(8) VARYING INITIAL('MAFY'),
        C CHARACTER(3),
        D CHARACTER(5) VARYING;
C = A;                               /* C is 'XZ/'    */
C = 'X';                             /* C is 'Xbb'    */
D = B;                               /* D is 'MAFY'   */
D = SUBSTR(A,2,3) !! SUBSTR(A,2,3); /* D is 'Z/BZ/'  */
SUBSTR(A,2,4) = B;                   /* A is 'XMAFY'  */
SUBSTR(B,2,2) = 'R';                 /* B is 'MRbY'   */
SUBSTR(B,2) = 'R';                  /* B is 'MRbb'   */

```

The BEGIN statement

Function:

The BEGIN statement is the heading statement of a begin block (see Chapter 2 for a discussion of blocks).

General format:

```
begin-statement ::= BEGIN;
```

General rules:

1. The BEGIN statement is used in conjunction with an END statement.
2. A begin block may not directly contain an ENTRY statement or a RETURN statement.

The CALL statement

Function:

The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of a procedure.

General format:

31 MARCH 1972

call-statement ::=

CALL entry-expression [(argument [, argument...])];

Syntax rules:

1. The entry-expression specifies the entry point of the invoked procedure.
2. An argument is an expression.

General rules:

1. The entry-expression can either be an entry constant or an entry variable that has had an entry value assigned.
2. Any argument expressions are evaluated when the CALL statement is executed. This includes the execution of any on-units entered as the result of conditions raised during the evaluation.
3. The called procedure is invoked in the environment of the entry value. If the containing block is inactive, the results are undefined. This value is established after the evaluation of the argument expressions and thus reflects any modifications made to the calling block's environment during the evaluation of the argument expressions.
4. The attributes of argument expressions must match the attributes of corresponding parameters. For details of the correspondence between arguments see "Correspondence of Arguments and Parameters" in Chapter 2.

31 MARCH 1972

Example:

```
A: PROCEDURE;  
    DECLARE X FIXED;  
B: PROCEDURE(I) RETURNS (FIXED);  
    DECLARE I FIXED;  
    X = 2;  
    RETURN (I+1);  
END B;  
X = 1;  
L: CALL C(B(5));  
    ...  
C: PROCEDURE(J);  
    DECLARE J FIXED;  
    ...
```

When procedure C is invoked at statement L, J will take on a value of 6. X will have the value 2. This occurs because the argument list of the CALL of C causes an invocation of B as a function. The function B sets the variable X declared in A to the value 2 and returns a value one greater than I, namely 6.

The CREATE statement

(see the ALLOCATE statement)

The DECLARE statement

Function:

The DECLARE statement is a non-executable statement used in the specification of attributes of simple names. Attributes common to several names can be factored to eliminate repeated specification of the same attribute for many identifiers. This factoring is achieved by enclosing the same declarations in parentheses and following this by the set of attributes to be applied. Level numbers, for structure declarations, may also be factored, but in such cases, the level number precedes the parenthesized list of names.

31 MARCH 1972

General format:

declare-statement ::=

DECLARE declaration-list;

declaration-list ::=

declaration [, declaration] ...

declaration ::=

$$\left\{ \begin{array}{l} \text{integer}(\text{simple-declaration-list}) \\ ((\text{declaration-list}) \\ ([\text{integer}] \text{identifier}) \end{array} \right\} \text{attribute-list}$$

simple-declaration-list ::=

simple-declaration [, simple-declaration] ...

simple-declaration ::=

$$\left\{ \begin{array}{l} \text{identifier} \\ (\text{simple-declaration-list}) \end{array} \right\} \text{attribute-list}$$

attribute-list ::=

[(dimension-attribute)] [attribute ...]

Syntax rules:

1. Any number of identifiers may be declared as names in one DECLARE statement.
2. Attributes must follow the names to which they refer.
3. The "integer" indicates the level in a structure declaration and must be an unsigned decimal integer greater than zero. If it is not specified, level 1 is assumed. All structure declarations must be preceded by a level number.
4. A DECLARE statement may have a label prefix. On transfer of control to such a label, the label is treated as if it were on a null statement and execution continues with the next executable statement.

31 MARCH 1972

General rules:

1. A data type attribute must be specified for all scalars, arrays of scalars, and structure elements.
2. All of the attributes given explicitly for a particular name must be declared together in one DECLARE statement.
3. No attribute may be specified more than once for the same name.
4. Attributes of EXTERNAL names, declared in separate blocks, must not conflict or supply explicit information that was not explicit or implicit in other declarations.

Declaration of structures:

The outermost structure is a major structure and all contained structures are minor structures. A structure is specified by declaring the major structure name and following it with the names of all contained elements. Each name is preceded by a level number as defined in the syntax rules. A major structure is always at level one and all elements contained in a structure (at level n) have a level number that is numerically greater than n , but they need not necessarily be a level $n+1$, nor need they all have the same level number.

A minor structure at level n contains all following items declared with level numbers greater than n up to but not including the next item with a level number less than or equal to n . A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of the DECLARE statement.

31 MARCH 1972

Examples:

```

DECLARE ((A FIXED,
          B FLOAT) STATIC,
          C ENTRY) EXTERNAL;

```

This declaration is equivalent to the following:

```

DECLARE A FIXED STATIC EXTERNAL,
        B FLOAT STATIC EXTERNAL,
        C ENTRY EXTERNAL;

```

```

DECLARE 1 S AUTOMATIC,
        2 (T FIXED,
           U FLOAT,
           V CHARACTER(10));

```

This declaration is equivalent to the following:

```

DECLARE 1 S AUTOMATIC,
        2 T FIXED,
        2 U FLOAT,
        2 V CHARACTER(10);

```

The DELETE statement

(see the FREE statement)

The DO statement**Function:**

The DO statement delimits the start of a do-group and may specify repetitive or selective execution of the statements within the group.

General format:

do-statement ::=

```

DO [ WHILE(relational-expression)
    CASE(scalar-integer-expression)
    variable = specification ]

```

31 MARCH 1972

specification ::=

$$\text{expr1} \left[\begin{array}{l} \text{TO expr2 [BY expr3]} \\ \text{BY expr3 [TO expr2]} \end{array} \right] [\text{WHILE}(\text{relational-expression})]$$

Syntax rules:

1. The "variable" is a scalar arithmetic variable of any storage class.
2. Each "expr" in the specification is a scalar expression.
3. If the BY clause is omitted from the specification and the TO clause appears, the value of expr3 is assumed to be 1.
4. If the TO clause is omitted from the specification and the BY clause appears, the iteration is performed until termination by the WHILE clause, if present, or by some other statement within the group.
5. If both the TO and BY clauses are omitted, this form of the specification implies a single execution of the do-group with the control variable having the value of expr1 or it implies no execution if the WHILE statement is false.

General rules:

1. In a simple DO statement without any iterative, relational, or selective specification, the statement serves to delimit the start of a do-group.
2. If only a WHILE clause is specified, the DO statement delimits the start of a do-group and specifies repetitive execution defined by the following:

31 MARCH 1972

```

LABEL: DO WHILE ( relational-expression );
        statement-1
        ...
        statement-n
      END;
NEXT: statement

```

The above is exactly equivalent to the following expansion:

```

LABEL: IF ~(relational-expression) THEN
        GO TO NEXT;
        statement-1
        ...
        statement-n
        GO TO LABEL
NEXT: statement

```

3. If a CASE clause is specified, the DO statement delimits the start of a do-group and specifies that a particular statement of the group is to be executed. Following execution of the selected statement, control passes to the statement following the group unless the executed statement causes a transfer of control. A statement in this context may be a single statement, a do-group, or a BEGIN block. The execution of the DO CASE group is defined as follows:

```

        DO CASE (scalar-integer-expression);
            statement-0
            ...
            statement-n
        END;
NEXT: statement

```

The above is exactly equivalent to the following expansion:

```

        DECLARE L(0:n) LABEL CONSTANT(L0,L1,...Ln);
        GO TO L(scalar-integer-expression);
L0: statement-0
    GO TO NEXT;
L1: statement-1
    GO TO NEXT;
    ...
Ln: statement-n
NEXT: statement

```

31 MARCH 1972

If the value of the scalar-integer-expression is outside the range 0 to n then the program is in error and the results are undefined.

4. If the DO statement defines a variable and a specification, the statement delimits the start of a do-group and specifies controlled repetitive execution defined by the following:

```
DO variable(a1,...,am) = expr1 TO expr2
  BY expr3 WHILE(expr4);
  statement-1
  ...
  statement-n
```

```
LABEL1: END;
NEXT: statement
```

This is exactly equivalent to the following expansion:

```
temp1 = a1;
...
tempm = am;
e1 = expr1;
e2 = expr2;
e3 = expr3;
v = e1;
LABEL2: IF(e3>=0) & (v>e2) ! (e3<0) & (v<e2) THEN
GO TO NEXT;
IF (expr4) THEN;
ELSE GO TO NEXT;
statement-1
...
statement-n
LABEL1: v = v + e3;
GO TO LABEL2;
NEXT: statement
```

In the above expansion, a_1, \dots, a_n are expressions that may appear as subscripts of the control variable, and $temp_1, \dots, temp_n$ are compiler-created integer variables to which the expression values are assigned; v is equivalent to "variable" with the associated "temp" subscripts; "e1", "e2", and "e3" are compiler-created variables having the attributes of "expr1", "expr2", and "expr3" respectively. In the simplest cases, there are no subscripts (i.e. $m = 0$) and the first statement in the expansion is therefore: $e_1 = expr_1$;. Additional rules for the above expansion follow:

31 MARCH 1972

- a. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in the expansion is omitted.
 - b. If "TO expr2" is omitted, the statement "e2 = expr2;" and the IF statement identified by LABEL2 are omitted.
 - c. If both "TO expr2" and "BY expr3" are omitted, all statements involving e2 and e3 as well as the statement GO TO LABEL2, are omitted.
 - d. Although the above expansions show a specific order in which the BY and TO clauses are evaluated, this order is undefined.
5. The WHILE clause specifies that before each associated execution of the do-group, the relational-expression is evaluated and, if the result is false, the iterations associated with the current iteration are terminated.
 6. In the specification, expr1 represents the starting value of the control variable. Expr3 represents the increment to be added after each iteration to the control variable. Expr2 represents the terminating value of the control variable. Iteration terminates as soon as the value of the control variable passes its terminating value. When the last specification is completed, control passes to the statement following the do-group.
 7. Control may, under any circumstances, be transferred into a do-group from outside the do-group provided that no iteration or selection is specified on the DO statement that delimits the group. If the do-group is selective or iterative, a GO TO statement can transfer control to a statement inside the group if the GO TO specifies an out-of-block transfer from a block that has been activated from within the do-group.
 8. The effect of allocating or freeing the control variable is undefined.

31 MARCH 1972

Examples:

```
DO INDEX = Z WHILE(A < B);
DO I = 1 TO 9;
DO CASE(3*I+5);
DO;
DO WHILE(TAX - DEDCT > ESTTAX * 4);
```

The END statement**Function:**

The END statement terminates blocks and groups.

General format:

```
end-statement ::= END [identifier] ;
```

General rules:

1. The END statement terminates that group or block headed by the nearest preceding DO, BEGIN, PROCEDURE, or FOR EACH statement for which there is no other corresponding END statement.
2. If an identifier follows the END, the block or group closed by the END statement must be preceded by the same label.
3. If control reaches an END statement terminating a procedure it is treated as a RETURN statement.
4. If control reaches an END statement terminating a begin block that is an on-unit, control is returned to the point specified for that particular interrupt. This is a "normal return" from the on-unit.

The ENTRY statement**Function:**

The ENTRY statement specifies a secondary entry point to a procedure.

31 MARCH 1972

General format:

entry-statement ::=

```

    entry-name: ENTRY[ (parameter [, parameter]...) ]
    [ RETURNS (data-attributes) ];

```

General rules:

1. Each "parameter" identifies a variable that is to be received at the specified entry point. When the entry is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point.
2. If the entry is invoked as a function reference, the RETURNS option must be specified. The data-attributes of the RETURNS option specify the attributes of the value returned by the entry. The attributes that may be specified are the arithmetic, string, locator, and file attributes.
3. An ENTRY statement cannot be internal to a begin block, nor can it be internal to a group that specifies iteration or selection.

The EXIT statement

Function:

The EXIT statement causes immediate termination of the program that contains the statement.

General format:

```

    exit-statement ::= EXIT;

```

General rule:

If an EXIT statement is executed, the FINISH condition is raised. On normal return from the FINISH on-unit, the program is terminated.

31 MARCH 1972

The FIND Statement

Function:

The FIND statement is used to locate a specified entity that is a member of a set or container of a set.

General format:

find-statement ::=

```
FIND find-specification [[,] ELSE statement ] ;
```

find-specification ::=

```
entity-specification-1 [= [ (integer-expression) ] ]
find-definition
```

find-definition ::=

```
{ ENTITY contain-clause }
{ entity-identifier contain-clause exception-clause }
```

contain-clause ::=

```
{ IN set-definition [ FROM entity-specification-2 ] }
{ CONTAINING entity-specification-2 }
{ IN entity-specification-1 -> set-name }
```

exception-clause ::=

```
[[,] WITH relational-expression-1 ]
[[,] UNTIL relational-expression-2 ]
```

entity-specification ::=

```
{ locator-variable }
{ entity-variable }
```

set-definition ::=

```
{ locator-variable }
{ set-name [ OF (file-variable) ] }
{ character-string-expression [ OF (file-variable) ] }
{ locator-variable -> set-name }
{ locator-variable -> (character-string-expression) }
```

31 MARCH 1972

General rules:

1. In the following rules, the value of the scalar-integer-expression will be referred to as "n". If the integer is not specified, a constant integer value of one is assumed.
2. The FIND statement searches the set referenced in the set-definition for the n-th entity that satisfies the conditions defined in the "exception-clause" part of the FIND statement. The locator or entity variable named in "entity-specification-1" is set to reference this entity.
3. If n is positive, the direction of search of the set is from the first entity to the last entity. If n is negative, the search is in the opposite direction. The 0-th member of a set is the entity that contains the set.
4. If the FROM option is specified, the search starts from the entity referenced in "entity-specification-2" and proceeds in the direction defined in rule 3.
5. If the FROM option is omitted, the search starts at the first entity and proceeds in the direction defined in rule 3.
6. The search is terminated when either the required entity has been found, a successful search, or when the entity containing the set is encountered in the course of the search before ABS (n) entities that satisfy the specified conditions have been found, an unsuccessful search.
7. If the search is unsuccessful, "entity-specification-1" will be set to reference the entity that contains the set unless the "set-definition" references a file-set, in which case entity-specification-1 will be set to the value of the NULL built-in function. If the optional ELSE clause has been specified, the statement following the keyword ELSE will be executed. If no ELSE clause is specified, the FIND condition will be raised.

31 MARCH 1972

8. If the keyword ENTITY is specified, all entities, regardless of name, are examined.
9. If the keyword ENTITY is not specified, only entities that are generations of the entity specified by "entity-identifier" are examined.
10. If the optional WITH clause is specified, the relational expression is evaluated for each entity examined, and the entity is only counted in the search if the relational expression yields the value true.
11. If the optional UNTIL clause is specified, the relational expression is evaluated for each entity examined and the search is terminated if the relational expression yields the value true. In this case, entity-specification-1 will be set to reference the entity on which the search terminated.
12. If a locator variable is used as the set-definition, its value must have been set by the LET statement. The character-string-expression specified in the set-definition must be the name of a defined set. If the character-string-expression is itself a qualified based variable, it must be enclosed in parentheses.
13. The keyword IN is synonymous with the keyword ON in the FIND statement.
14. If the CONTAINING clause is specified, a search will be made for the n-th entity that contains the set and member entity referenced by entity-specification-2. The order of search (for positive values of n) corresponds to the order in which the reference entity was inserted onto different sets. Entity-Specification-1 must appear twice in the FIND statement; first as the unknown and second as the identifier of the class of sets to be searched.
15. If the search for a containing entity is unsuccessful, the optional ELSE clause will be executed with entity-specification-1 set equal to entity-specification-2. If the ELSE clause is not present, the FIND condition is raised.

31 MARCH 1972

16. The FROM clause cannot be used with the CONTAINING option. If the locator variable within a FROM clause has the value NULL, the search will begin with the first member of the set.
17. If a set-name or character-string-expression is used by itself in the set-definition, a FILE_SET in the specified file (or the default "current" file) will be referenced.

Examples:

```
FIND P1 = (1) ENTITY IN P2 -> SETA FROM P3 ;
FIND PTR = (J) LINE ON BNDRY ELSE GO TO ERR ;
FIND POINT = (1) ENTITY
CONTAINING P -> POINT IN POINT -> PSET ;
```

The FOR EACH statement**Function:**

The FOR EACH statement delimits the start of a group and defines the repetitive execution of the statements within the group.

General format:

```
for-each-statement ::= FOR EACH find-specification ;
```

Syntax rule:

The syntax of the "find-specification" is defined in "The FIND statement" in this chapter.

General rules:

1. The FOR EACH statement is a means for the application of an algorithm to all or selected members of a set or entities that contain a set. The scope of the FOR EACH statement is terminated by the END statement and all the rules applicable to an iterative do-group are also applicable to a for-each group.

31 MARCH 1972

2. The effect of a FOR EACH statement is defined by the following:

```

FOR EACH find-specification ;
    statement-1
    ...
    statement-n
END;
NEXT: statement

```

The above FOR EACH group is exactly equivalent to the following:

```

ptemp = fromp;
DO WHILE('1'B);
    FIND find-specification FROM ptemp ELSE
    GO TO NEXT;
    statement-1
    ...
    statement-n
    ptemp = pfind;
END;
NEXT: statement

```

where ptemp is a compiler defined pointer variable, fromp is the value of the entity-specification-2 in the FROM clause or, if the FROM clause is omitted, is the value of the NULL built-in function, pfind is the locator-variable or entity-variable specified in entity-specification-1.

Examples:

```

FOR EACH P1 = ENTITY ON S1;
    A = A + P1 -> B ;
END ;

```

is equivalent to:

```

P = NULL ;
DO WHILE('1'B);
    FIND P1 = ENTITY ON S1 FROM P
    ELSE GO TO DONE ;
    A = A + P1 -> B ;
    P = P1 ;
END;
DONE :

```

31 MARCH 1972

```

FOR EACH P1 = (2) LINE IN 'SETA' FROM P2
    WITH P1 -> LINE.X < 0 ;
    P1 -> LINE.X = 0 ;
END;

```

is equivalent to:

```

P = P2 ;
DO WHILE('1'B);
    FIND P1 = (2) LINE IN 'SETA' FROM P
        WITH P1 -> LINE.X < 0
        ELSE GO TO DONE;
    P1 -> LINE.X = 0 ;
    P = P1 ;
END;
DONE :

```

The FREE statement

Function:

The FREE statement causes the storage allocated for specified based variables to be freed.

General format:

free-statement ::=

```
FREE free-specification [,free-specification...];
```

free-specification ::=

```
[locator-variable ->] based-variable
[ INCLUSIVE ] [ IN(scalar-file-variable) ]
```

Syntax rule:

The "based-variable" must be an unsubscripted level-1 based variable.

General rules:

1. A based variable can be used to free storage only if that storage has been allocated for a based variable having identical data attributes, including values of bounds and lengths.

31 MARCH 1972

2. An IN option must be specified if the generation to be freed was allocated in a file. It may not be specified if the generation to be freed was allocated in scratch storage. The IN option must specify the file in which the generation was allocated.
3. The effect of the FREE statement is to make the specified storage available for subsequent allocation by an ALLOCATE statement.
4. If the reference to the variable to be freed is pointer-qualified by the POINTER built-in function (either explicitly, or implicitly by the appearance of an offset as the locator qualifier), and the IN option is absent, the statement is executed as if it contains the IN option naming the file that is the second argument of the POINTER built-in function.
5. If the storage to be freed has been allocated in scratch storage, as opposed to a particular file, the FREE statement cannot include an IN option nor can an IN option be implied by the use of an offset as a locator qualifier.
6. The FREE statement may be used to free the storage space for ENTITY variables. An entity that is freed will be removed from all sets of which it is a member. Then the entity is freed. If the entity contains other sets, the member entities of these sets will be freed provided they are members of no other sets. If the INCLUSIVE option is used, all member entities will be freed regardless of their membership in other sets. This process continues recursively until no more entities can be freed.

Example:

```
DECLARE F FILE,  
        Q OFFSET (F),  
        V BASED (Q);  
FREE V;
```

The FREE statement is equivalent to the statement:

```
FREE POINTER(Q, F) -> V IN(F);
```

31 MARCH 1972

The GET Statement

Function:

The GET statement causes values, either from a sequential file or from a string variable, to be assigned to variables specified in a data list.

General format:

```
get-statement ::= GET get-list ;
```

```
get-list ::=
```

$$\left. \begin{array}{l} \text{FILE(file-variable)} \\ \text{STRING(character-string-variable)} \end{array} \right\} \text{data-specification}$$

General rules:

1. The file-variable must refer to a sequential file that has been opened.
2. The "character-string-variable" refers to the fixed length character string that is to provide the data to be assigned to the data list. Each GET operation using this option always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters implied by the data specification, the ERROR condition is raised.
3. The rules concerning the "data-specification" are defined in "Data Lists" in Chapter 6.

The GO TO statement

Function:

The GO TO statement causes control to be transferred to a statement identified by a label prefix.

General format:

31 MARCH 1972

$$\text{go-to-statement} ::= \left\{ \begin{array}{l} \text{GO TO} \\ \text{GOTO} \end{array} \right\} \left\{ \begin{array}{l} \text{label-constant} \\ \text{scalar-label-variable} \end{array} \right\};$$

General rules:

1. If a label variable is specified, the GO TO statement has the effect of a multi-way switch. The value of the label variable is the label of the statement to which control is transferred. Since the label variable may have different values at each execution of the GO TO statement, control may not always pass to the same statement.
2. A GO TO statement cannot pass control to an inactive block.
3. A GO TO statement cannot transfer control from outside a group to a statement inside the group if the group specifies iteration or selection except in the case where the GO TO specifies an abnormal return from a block that has been invoked from within the group.
4. A GO TO statement that transfers control from one block, D, to a dynamically encompassing block, A has the effect of terminating block D, as well as all other blocks that are dynamically descendant from block A. On-units are reestablished and automatic variables are freed in the same way as if the blocks were terminated normally. When a GO TO statement transfers control out of a procedure invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued. The value returned by the procedure being terminated is undefined, and control is transferred to the specified statement.
5. A GO TO cannot terminate any block activated during the execution of an ALLOCATE statement.

Examples:

```

        GO TO A2345;
        ...
A2345:  ...

```

The following example illustrates a GO TO statement that acts as a multi-way switch:

31 MARCH 1972

```

        DECLARE L LABEL INITIAL (L2);
        GO TO MEET;
L1:    X = Y - 1;
        L = L2;
        GO TO MEET;
L2:    Y = X - 1;
        L = L1;
MEET:  CALL FUDGE(X,Y,Z);
        IF Z = LIMIT THEN
        GO TO L;
        ...

```

The following procedure illustrates the use of the GO TO statement with a subscripted label variable to effect a multi-way switch:

```

        DECLARE (N1, N2) FIXED,
                SWITCH(3) LABEL;
        SWITCH(1) = CALC1;
        SWITCH(2) = CALC2;
        SWITCH(3) = CALC3;
        GO TO SWITCH(MOD(N1 + N2, 3) + 1));
        ...
CALC1: ...
        ...
CALC2: ...
        ...
CALC3: ...
        ...

```

The IF statement

Function:

The IF statement specifies evaluation of a relational expression and a consequent flow of control dependent upon the truth value of the expression.

General format:

```

if-statement ::= IF scalar-relational-expression
                THEN then-clause
                [ELSE else-clause]

```

Syntax rules:

31 MARCH 1972

1. Each then-clause and else-clause is a group, a begin-block, or any statement other than DECLARE, END, ENTRY, or PROCEDURE. The unit may have its own labels.
2. The IF statement is not itself terminated by a semicolon.

General rules:

1. The scalar-relational-expression is evaluated, then:
 - a. If the value of the expression is true, the then-clause is executed and control is passed to the statement following the IF statement.
 - b. If the value of the expression is false and an else-clause is specified, then the else-clause is executed and control is passed to the statement following the IF statement.
 - c. If the value of the expression is false and an else-clause is not specified, control is passed to the statement following the IF statement.
2. Either the then-clause or the else-clause may contain GO TO statements that transfer control to statements outside the IF statement. If such a GO TO statement is executed, control will not be passed to the statement following the IF statement.
3. IF statements may be nested, that is, either the then-clause or the else-clause, or both, may themselves be IF statements. Each ELSE clause is always associated with the innermost unmatched IF in the same block or do-group. As a consequence, an ELSE or a THEN with a unit consisting of a null statement may be required to specify a desired sequence of control.

31 MARCH 1972

Examples:

```

IF A + B = Z THEN CALL X(0);
                    ELSE CALL X(A);

```

```

IF X < Y THEN
    IF Z = W THEN
L:     Y = 1;
    ELSE;
ELSE
    Y = A;

```

```

IF A THEN
    GO TO M;
GO TO N;

```

The INSERT statement**Function:**

The INSERT statement causes a referenced entity to be inserted on a specified set.

General format:

insert-statement ::=

INSERT entity-specification-1 IN set-definition

```

[ FIRST
  LAST
  BEFORE entity-specification-2;
  AFTER entity-specification-2 ]

```

entity-specification ::=

```

{ locator-variable }
{ entity-variable }

```

set-definition ::=

```

{ locator-variable
  set-name [ OF(file-variable) ]
  character-string-expression [ OF(file-variable) ]
  locator-variable -> set-name
  locator-variable -> (character-string-expression) }

```

31 MARCH 1972

General rules:

1. The `INSERT` statement makes the entity referenced by entity-specification-1 a member of the specified set. If the optional `FIRST`, `LAST`, `BEFORE`, or `AFTER` clause is omitted, `LAST` will be assumed.
2. If the `BEFORE` or `AFTER` clauses are used, the entity referred to by entity-specification-2 must be a member of the specified set at the time the `INSERT` statement is executed. If this entity cannot be located, the `FIND` condition will be raised.
3. The character-string-expression specified by the set will be truncated to 8 characters if necessary. If the expression itself is a qualified based character string, it must be enclosed in parentheses.
4. If a locator variable is used as the set definition, the locator must reference an existing set. The `LET` statement can be used to set a locator-variable to reference a set.
5. The keyword `IN` is synonymous with `ON` within the `INSERT` statement.
6. The member `ENTITY` and `SET` in which it is to be inserted, must be contained in the same file.
7. If a set-name or character-string-expression is used by itself in the set-definition, a `FILE_SET` in the specified file (or the default "current" file) will be referenced.

The LET statement**Function:**

The `LET` statement sets a locator variable to reference a specified set.

31 MARCH 1972

General format:

let-statement ::= LET locator-variable = set-definition ;

set-definition ::=

$$\left\{ \begin{array}{l} \text{locator-variable} \\ \text{set-name [OF(file-variable)]} \\ \text{character-string-expression [OF(file-variable)]} \\ \text{locator-variable} \rightarrow \text{set-name} \\ \text{locator-variable} \rightarrow (\text{character-string-expression}) \end{array} \right\}$$

General rules:

1. The character-string-expression specified by the set definition will be truncated to 8 characters if necessary. If the expression itself is a qualified based character string, it must be enclosed in parentheses.
2. If a set-name or character-string-expression is used by itself in the set-definition, a FILE_SET in the specified file (or the default "current" file) will be referenced.

The LOCK statement

Function:

The execution of the LOCK statement puts the program into locked status.

General format:

lock-statement :: = LOCK ;

General rules:

1. When a program is in locked status, all asynchronous events will be queued. No on-units specified in ON EVENT statements will be invoked while a program is in locked status.
2. On-units established for system conditions or programmer defined conditions are not affected by the locked or unlocked status of program.

31 MARCH 1972

3. The execution of a LOCK statement while in the locked status is equivalent to a null statement.
4. A program will remain in locked status until explicitly unlocked or until control reverts to a dynamically encompassing block in which the status is unlocked.

The null statement

Function:

The null statement is a no-operation.

General format:

null-statement ::= ;

Example:

```

...
ON OVERFLOW;
...

```

The overflow on-unit is a null statement.

The ON statement

Function:

The ON statement specifies the action to be taken when an interrupt occurs for the named condition or non-delayed event. For a discussion of conditions and events, see the description of "Interrupt Handling", Chapter 7.

General format:

on-statement ::=

```

ON [ EVENT ] identifier [, identifier ]... { (on-unit)
                                           (SYSTEM) } ;

```

31 MARCH 1972

Syntax rules:

1. If the keyword `EVENT` is omitted, the "identifier" must be the name of one of the conditions described in Appendix 2.
2. The "on-unit" is an action specification, and it is either an unlabeled single simple statement (other than `BEGIN`, `DO`, `END`, `RETURN`, `ENTRY`, `PROCEDURE`, `FOR EACH`, or `DECLARE`) or an unlabeled begin block. Since the on-unit itself requires a semi-colon, none appears in the format.
3. The "on-unit" may not be a `RETURN` statement, nor may a `RETURN` statement be internal to the begin block.
4. If the keyword `EVENT` is present, the `ON` statement must be within the scope of a declaration of the identifier as an `EVENT`.
5. The specification of more than one identifier is equivalent to the specification of identical actions for each named interrupt.

General rules:

1. An `ON` statement must be executed before its effect can be established.
2. The standard action to be taken for all interrupts is defined in Appendix 2. When an interrupt takes place before an `ON` statement for that interrupt has been executed, standard system action is taken. The `ON` statement with the `SYSTEM` option specifies that standard action is to be taken when the named interrupt occurs.
3. The `ON` statement is a means for the programmer to specify action (other than standard system action) that is to take place when the named interrupt occurs. The on-unit is treated as a block that is internal to the block in which it appears.
4. Control can reach an on-unit only when the named interrupt occurs, or when a `SIGNAL` statement for the interrupt is executed.
5. If an action specification is established by execution of an `ON` statement, it remains in effect

31 MARCH 1972

until it is overridden by another ON statement or REVERT statement specifying the same interrupt, or until termination of the block in which the ON statement is executed.

The PROCEDURE statement

Function:

The PROCEDURE statement has the following functions:

1. Identifies a portion of program text as a procedure.
2. Defines the primary entry point to a procedure.
3. Specifies the parameters for the primary entry point.
4. Specifies the attributes of the value that is returned if the procedure is invoked as a function at the primary entry point.

General format:

```
procedure-statement ::= entry-name: PROCEDURE
                        [(parameter [, parameter] ...)]
                        [RETURNS(data-attributes)] ;
```

General rules:

1. Each "parameter" is a name that specifies the parameters of the entry point. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point (see "Correspondence of Arguments and Parameters" in Chapter 2.)
2. If the entry is invoked as a function reference, the RETURNS option must be specified. The data-attributes of the RETURNS option specify the attributes of the value returned by the entry. The attributes that may be specified are the arithmetic, string, locator, and file attributes.

31 MARCH 1972

The PUT statement**Function:**

The PUT statement causes the values of expressions to be converted to a character string representation according to specified formats and to be transmitted to a designated sequential file or string variable.

General format:

```
put-statement ::= PUT put-list ;
```

```
put-list ::=
```

```

{ FILE(file-variable)
  STRING(character-string-variable) } data-specification

```

General rules:

1. The "character-string-variable" refers to the character string variable that is to receive the transmission. After appropriate conversion, the data specified in the data-specification is assigned to the string starting at the leftmost character. Any subsequent PUT statement naming the same string will start assigning at the leftmost character. If the string is not long enough to accommodate the data, the ERROR condition will be raised.
2. The "file-variable" must refer to a sequential file that has been opened.
3. The rules concerning the "data-specification" are contained in "Data Lists" in Chapter 6.

The REMOVE statement**Function:**

The REMOVE statement is used to remove an entity from a set.

31 MARCH 1972

General format:

remove-statement ::=

```
REMOVE entity-specification FROM {set-definition};
                                { all-value }
```

entity-specification ::=

```
{locator-variable}
{entity-variable }
```

set-definition ::=

```
{ locator-variable
  set-name [ OF(file-variable) ]
  character-string-expression [ OF(file-variable) ]
  locator-variable -> set-name
  locator-variable -> (character-string-expression) }
```

General rules:

1. The character expression in the set-definition is evaluated and truncated to 8 characters if necessary. If the expression itself is a qualified based character string, it must be enclosed in parentheses.
2. If the referenced entity is not a member of the specified set the REMOVE statement has the effect of a null statement.
3. The "all-value" is the value returned by the ALL built-in function which, if used, must be declared with the BUILTIN attribute. The entity will then be removed from all sets that contain it.
4. If a locator variable is used to specify a set, the locator variable must reference an existing set. The LET statement must be used to set a locator variable to reference a set.
5. If a set-name or character-string-expression is used by itself in the set-definition, a FILE-SET in the specified file (or the default "current" file) will be referenced.

31 MARCH 1972

The RETURN statement

Function:

The RETURN statement terminates execution of the procedure that contains the RETURN statement and returns control to the invoking procedure. The RETURN statement may also return a value.

General format:

```
return-statement ::= RETURN [ (scalar-expression) ];
```

General rules:

1. If the procedure is not invoked as a function procedure, i.e., it has been invoked by a CALL statement, the RETURN statement may specify a scalar expression, but the value will be ignored by the invoking procedure.
2. If the procedure is invoked as a function procedure, the RETURN statement used to terminate the procedure must specify a value that is to be returned to the invoking procedure by specifying a scalar expression. There is no type conversion implied by the RETURNS attribute on the PROCEDURE or ENTRY statement.
3. If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement that does not specify a value to be returned.
4. A RETURN statement may not be internal to a begin-block.

The REVERT statement

Function:

A REVERT statement specifying a given condition or event is used to cancel the effect of one or more previously executed ON statements.

31 MARCH 1972

General format:

revert-statement ::=

REVERT [EVENT] identifier [, identifier] ... ;

Syntax rules:

1. If the keyword EVENT is omitted, the "identifier" must be the name of one of the conditions described in Appendix 2, or the name of a user defined condition appearing in a DECLARE statement.
2. If the keyword EVENT is present, the REVERT statement must be contained within the scope of a declaration of the identifier as an EVENT.

General rule:

The execution of a REVERT statement has the effect described above only if (1) an ON statement, specifying the same conditions or events and internal to the same block, was executed after the block was activated and (2) the execution of no similar REVERT statement has intervened. If either of the two conditions is not met, the REVERT statement is treated as a null statement.

The SIGNAL statement

Function:

The SIGNAL statement simulates the occurrence of the named interrupts.

General format:

signal-statement ::=

signal cond-or-event[, cond-or-event] ... ;

cond-or-event ::=

$$\left. \begin{array}{l} \text{identifier} \\ \text{EVENT event-name [(locator-expression)]} \end{array} \right\}$$

31 MARCH 1972

Syntax rules:

1. If the keyword **EVENT** is omitted, the "identifier" must be the name of one of the conditions described in Appendix 2.
2. If the keyword **EVENT** is present, the **SIGNAL** statement must be contained within the scope of the declaration of the identifier as an **EVENT**.

General rules:

1. The **SIGNAL** statement with the **EVENT** option can be used to simulate the occurrence of the external interrupt associated with the event-name. The event will be set complete and if it is in the delayed state, the occurrence will be added to the event queue. If the event is non-delayed, the associated on-unit will be entered, (see "Interrupt Handling" in Chapter 7).
2. The optional locator-expression associated with an event-name may be used to supply the block of status information associated with the event and will be made available to the program handling the interrupt through the use of the **ONPTR** built-in function. If the locator-expression is omitted, the value of the corresponding invocation of the **ONPTR** built-in function will be the value of the **NULL** built-in function.

The UNLOCK statement**Function:**

The execution of the **UNLOCK** statement puts the program into unlocked status.

General format:

```
unlock-statement ::= UNLOCK;
```

General rules:

31 MARCH 1972

1. When a program is in the unlocked status, asynchronous events may be processed. For details on the processing of asynchronous events see "Interrupt Handling" in Chapter 7.
2. The execution of an UNLOCK statement while the program is in the unlocked status has the same effect as the execution of a null statement.
3. A program will remain in unlocked status until explicitly locked, or until control reverts to a dynamically encompassing block in which the status is locked.

The WAIT statement

Function:

The WAIT statement is used to synchronize the processing of delayed event completions. The terms "delayed" and "completion" are defined in Chapter 7 -- "Interrupt Handling" .

General format:

wait-statement ::=

$$\text{WAIT} \left\{ \begin{array}{l} ((\text{event} [, \text{event}] \dots) [\text{SET}(\text{fixed-scalar-variable})]) \\ (\text{ANY}) \end{array} \right\};$$

General rules:

1. The items specified in the list may be any delayed events. If any of the events are non-delayed, the ERROR condition will be raised.
2. The ANY option specifies the set of all delayed events. If the set is empty, the ERROR condition will be raised.
3. The WAIT statement is satisfied and execution proceeds to the statement following the WAIT statement when at least one of the specified events becomes complete. If the SET option is used, a fixed-scalar-variable will be set equal to the index of the event that became complete. If none of the specified events is complete, execu-

31 MARCH 1972

tion is suspended until one of the specified events becomes complete.

4. If a non-delayed event becomes complete while in a wait state, the on-unit for that event will be entered. Upon normal return from the on-unit, the wait state will be resumed.
5. If a delayed event that is not specified in the event list for the WAIT statement becomes complete while in the wait state, no action is taken.

31 MARCH 1972

APPENDIX 1 - BUILT-IN FUNCTIONS, PROCEDURES, AND PSEUDO-VARIABLESINTRODUCTION

All of the built-in functions, built-in procedures, and pseudo-variables that may be invoked by the Apple programmer are listed in this appendix. Each function or pseudo-variable that has an argument list may be used without declaration, unless an identifier has been declared with the same name. In this case, the function or pseudo-variable must be redeclared using the BUILTIN attribute. Each built-in function or pseudo-variable that has no argument must be declared with the BUILTIN attribute.

The built-in functions, procedures, and pseudo-variables are separated into the following classes: arithmetic, array, associative data handling, conversion, interrupt handling, mathematical, storage management, string handling, and miscellaneous.

31 MARCH 1972

ARITHMETIC

ABS
CEIL
FLOOR
MOD

ARRAY

DIM
HBOUND
LBOUND

ASSOCIATIVE DATA HANDLING

APLESET
APLEVAR
APLINDX
APLNUMB
APLQWNI
APLOWRS
APLSNAM
APLTYPE
** ALL

CONVERSION

* BYTE
CHAP
ENTRY
FIXED
FLOAT
HEX
OFFSET
POINTER

INTERRUPT HANDLING

COMPLETION
* DELAY
** ONFILE
** ONLOC
ONPTR

MATHEMATICAL

ATAN
COS
LOG
SIN
SQRT
TAN

STORAGE MANAGEMENT

ADDR
DESCR
** NULL

STRING HANDLING

INDEX
LENGTH
*** RAL
* SUBSTR

MISCELLANEOUS

** DATE
INLINE
** TIME

- * - Also pseudo-variables
- ** - Must be declared with the BUILTIN attribute
- *** - May only be used as a pseudo-variable

31 MARCH 1972

ARITHMETIC FUNCTIONSABS(x)

The ABS function returns the absolute value of x. The argument x must be a scalar arithmetic quantity. The value returned by ABS has the same scale and precision as x.

CEIL(x)

The CEIL function returns the smallest integer that is greater than or equal to x. The argument x must be a scalar arithmetic quantity. The value returned by CEIL has the same scale and precision as the argument x.

FLOOR(x)

The FLOOR function returns the largest integer value that does not exceed x. The argument x must be a scalar arithmetic quantity. The value returned by FLOOR has the same scale and precision as x.

MOD(x, d)

The MOD function returns the remainder from the division of x by d. The arguments x and d must be scalar arithmetic quantities. The result has the same sign, scale, and precision as x.

ARRAY FUNCTIONSDIM(a, d)

The DIM function returns the current extent of the dth dimension of a. The argument d must be a unsigned fixed point constant. The argument a must be a reference to an array that has at least d dimensions. The value returned is an integer.

HBOUND(a, d)

The HBOUND function returns the current upper bound of the dth dimension of a. The argument s must be a unsigned fixed point constant. The argument a must be a reference to an array that has at least d dimensions. The value returned is an integer.

31 MARCH 1972

LBOUND(a, d)

The LBOUND function returns the current lower bound of the dth dimension of a. The argument d must be a unsigned fixed point constant. The argument a must be a reference to an array that has at least d dimensions. The value returned is an integer.

ASSOCIATIVE DATA FUNCTIONSALL

The function returns a value that has meaning in the following contexts:

- a. Remove the referenced entity from all sets of which it is a member when ALL is used in the REMOVE statement.
- b. Search all members of the referenced set when ALL is used as an argument to the APLINDX built-in function.
- c. Count all members of the referenced set when ALL is used as an argument to the APLNUMB built-in function.
- d. Count all entities that contain the referenced set when ALL is used as an argument to the APLOWRS built-in function.

If this function is used, it must be within the scope of declaration of the identifier ALL with the attribute BUILTIN.

APLESET(s)

The function returns a value of '1'B or '0'B dependent on whether or not the file-set s has ever been created in the file determined by the value of SYSFILE.CURRENT. The set-reference, s, must be a character-string-expression.

APLEVAR(s)

The function returns a pointer variable that identifies the entity that contains the set s. If the referenced set is a file-set, the value returned is the value of the NULL built-in function. The set reference must be a locator variable set by the LET statement.

31 MARCH 1972

APLINDX(e, s, c)

The function returns an integer whose value is the ordinal of the entity e in the set s. The entity reference must be a locator variable. The set reference must be a locator value that has been set by the LET statement. If c is a character-string-expression it must contain the name of those entities to be included in the search. If c is a reference to the built-in function ALL, then all members of the set will be included in the search. If the search is unsuccessful, the value returned is zero.

APLNUMB(s, c)

The APLNUMB function returns an integer whose value is a count of the number of entities that are members of the set s. The set reference, s, must be a locator value that has been set by the LET statement. If c is a character-string-expression, it must contain the name of those entities to be included in the count. If c is a reference to the built-in function ALL, every member of the set will be included.

APLOWNI(e, s, c)

The APLOWNI function returns an integer whose value is the index of the set s from amongst all the sets having the name c of which e is a member. The entity reference, e, must be a locator variable. The set reference, s, must be a locator value that has been set by the LET statement. The value of the character-string-expression, c, must be the name of the set s, else the ERROR condition will be raised. If the entity e is not a member of the set s, the value zero is returned.

APLOWRS(e, c, d)

The APLOWRS function returns an integer whose value is the number of entities named d that contain a set named c of which e is a member. The entity reference e must be a locator variable. The set reference c must be a character-string-expression whose value is the name of a set. If d is a character-string-expression, it contains the name of those entities to be included in the count. If d is a reference to the built-in function ALL, every entity will be included.

31 MARCH 1972

APLSNAM(e, i, j)

The APLSNAM function returns a character-string of length 8. If the value of j is 1, the return value is the name of the ith set of which the entity e is a member. If the value of j is 0, the value returned is the name of the ith set contained by the entity e. The entity reference e must be a locator variable. Both i and j are fixed binary values of precision(23, 0). If j has a value other than 0 or 1, the ERROR condition is raised. If the search is unsuccessful, the value returned is (8) ' '.

APLTYPE(e)

The function returns a character string of length 8. The value of the string is the name of the entity e. The entity reference e must be a locator variable.

CONVERSION FUNCTIONSBYTE(x[, i])

The BYTE function interprets the first operand, x, as a vector of bit strings of length 8, aligned on an 8-bit boundary and with a lower bound of 1. The second argument, i, is interpreted as a subscript specifying which element of the array is to be referenced. (If i is omitted, 1 is assumed.) The 8-bit bit-string is converted to a positive FIXED BINARY(47) (see Chapter 4, Type Conversion, 4. Bit-string to Arithmetic) value which is returned by the BYTE function. The BYTE pseudo-variable can be used to assign a fixed-point value to an unsigned 8-bit integer which is the ith element of the array defined on x. The range of i is limited to $\pm 2^5$.

31 MARCH 1972

CHAR(v[, l])

The CHAR function returns the scalar arithmetic, entry or file value, v, converted to a character string according to the following rules:

- a. If v is a fixed point value of decimal precision d, the value is converted according to the EDIT format F(d+1).
- b. If v is a floating point value of decimal precision d, the value is converted according to the EDIT format E(d+8, d-1).
- c. If v is an entry value, the result is a character string containing the entry name left aligned. If v is not an external entry point, the ERROR condition will be raised.
- d. If v is a file value, the result is a character string containing the name of the external data set. If the file value is not an opened file, the UNDEFINEDFILE condition will be raised.

The optional argument, l, if supplied, must be a positive integer constant. If l is specified, the value of CHAR is the character string formed by taking the rightmost l characters of the string formed by the above rules. Otherwise, the result of CHAR is the fixed length character string formed according to the above rules.

ENTRY(c)

The ENTRY function returns the entry value corresponding to the external entry point whose name is the value of the character-string-expression c. If there is no external entry point of name c, the ERROR condition will be raised.

31 MARCH 1972

FIXED(x[, p])

The FIXED function returns the value of the string or arithmetic expression, x, converted to fixed point. The optional argument, p, which must be an unsigned decimal integer constant, specifies the decimal precision of the result. If p is not specified, 15 is assumed. The rules for the conversion of x to a fixed point value are:

- a. If x is of arithmetic type, the process is as described in "Arithmetic Conversion" in Chapter 4.
- b. If x is a character string then the conversion is according to F format conversion as described in Chapter 6, Fixed-point Format Items.
- c. If x is a bit-string, the conversion takes place according to the rules in "Type Conversion -- 4. Bit-string to Arithmetic" in Chapter 4.

FLOAT(x[, p])

The FLOAT function returns the value of the character string or arithmetic expression, x, converted to floating point. The optional argument, p, which must be an unsigned decimal integer constant, specifies the decimal precision of the result. If p is not specified, 15 is assumed. The rules for the conversion of x to a fixed point value are:

- a. If x is of arithmetic type, the process is as described in "Arithmetic Conversion" in Chapter 4.
- b. If x is a character string, then the conversion is according to E format conversion as described in "Floating-point Format Items" in Chapter 6.

HEX(f[, i[, l]])

The HEX function returns a character string containing the hexadecimal equivalent of the argument f. If the optional arguments are omitted, the resulting character string will represent the full extent of f, however, if the extent of f is greater than 32,767 bytes, only the first 32,767 bytes will be represented in the result. The optional arguments i and l are used in the same way as the second and third arguments of the built-in function SUBSTR (q.v.) to select a sub-string of the result. Thus, HEX(f, i, l) is exactly equivalent to SUBSTR(HEX(f), i, l). Any misuse of the HEX built-in function will bring down a curse on the program.

31 MARCH 1972

OFFSET(p, f)

The OFFSET function returns the offset value that identifies the same generation in the file f as is identified by the locator expression p. The argument f must be a file variable. the result of OFFSET is undefined if p does not identify a generation in f.

POINTER(o, f)

The POINTER function returns the pointer value that identifies the same generation in the file f as is identified by the offset expression o. The argument f must be a file variable. The result of POINTER is undefined if o does not identify a generation in f.

INTERRUPT HANDLING FUNCTIONSCOMPLETION(e)

The COMPLETION function returns a bit value of '0'B or '1'B dependant upon whether the event e is incomplete or complete.

DELAY(e)

The DELAY function returns a bit value of '0'B or '1'B dependant upon whether the event e is in the non-delayed or delayed state. (DELAY may also be used as a pseudo-variable).

ONFILE

The ONFILE function returns a varying length character string giving the name of the file for which an ENDFILE, CONVERSION, OR ERROR condition has been raised. If the condition is not associated with a file, a null string is returned. If this function is used, it must be within the scope of a declaration of the identifier ONFILE with the attribute BUILTIN.

31 MARCH 1972

ONLOC

Whenever a condition is raised or non-delayed event is completed, reference to the ONLOC function will yield a varying length character string giving the name of the entry point to the procedure that was interrupted. The names of internal procedures are qualified by the names of the statically encompassing procedures. If the ONLOC function is used out of context, a null string is returned. If this function is used, it must be within the scope of a declaration of the identifier ONLOC with the attribute BUILTIN.

ONPTR(e)

The ONPTR function returns a pointer value that identifies the Event Completion Block that was associated with the event e when it became complete. Reference to e in this way also sets the event e to incomplete. Reference to ONPTR(e) when e is incomplete yields the null pointer value.

MATHEMATICAL FUNCTIONSATAN(x)

The function ATAN returns the principle value of the inverse tangent of the arithmetic expression x expressed in radians. The precision of the result is the precision of x.

COS(x)

The function COS returns the value of the cosine of the arithmetic expression x expressed in radians. The precision of the result is the precision of x.

LOG(x)

The function LOG returns the natural logarithm of the value of the arithmetic expression x. If the value of x is ≤ 0 , the ERROR condition is raised. The precision of the result is the precision of x.

SIN(x)

The function SIN returns the sine of the value of the arithmetic expression x expressed in radians. The precision of the result is the precision of x.

31 MARCH 1972

SQRT(x)

The function SQRT returns the positive square root of the value of the arithmetic expression x. If x is < 0, the ERROR condition will be raised. The precision of the result is the precision of x.

TAN(x)

The function TAN returns the tangent of the value of the arithmetic expression x expressed in radians. The precision of the result is the precision of x.

STORAGE MANAGEMENT FUNCTIONSADDR(v)

The ADDR function returns a pointer value that identifies the generation of the variable v.

DESCR(l, a)

The DESCR returns a descriptor value consisting of the value of the arithmetic expression l as the length and the value of the locator or arithmetic expression a as the pointer part.

FILE(g)

The FILE function returns a file value corresponding to the file in which the generation of the based variable referenced by g is allocated.

NULL

The NULL function returns the null pointer value. The null pointer value compares unequal with all pointer values that identify generations. Any use of this function must be within the scope of a declaration of the identifier NULL with the BUILTIN attribute.

31 MARCH 1972

STRING HANDLING FUNCTIONSINDEX(s, p)

The INDEX function searches the string s for the string pattern p. If the configuration is found, INDEX returns an integer giving the starting location of p in s. If more than one instance of p exists in s, the location of the first one found in a left-to-right search will be returned. If p does not exist in s or the length of either of the arguments is zero, the value 0 will be returned. Both s and p must be character string variables or expressions.

LENGTH(s)

The LENGTH function returns the length of the string s.

RAL(b)

The RAL pseudo-variable is used for assigning strings with right-hand alignment instead of the normal left-hand alignment. If the source string is shorter than the target string b, the source string will be extended on the left with blanks or zeros according to whether b is a character string or bit string. If the source string is longer than the target, it will be truncated on the left. The string b must be a fixed length string.

SUBSTR(s, i[, j])

The SUBSTR extracts a substring of user-defined length from the string s and returns it. The value of i specifies the starting point of the substring and j, if specified, represents the length of the substring. Both i and j must be arithmetic expressions and are converted to integers. Assuming that the length of s is k, then i and j must satisfy the following conditions:

- a. j must be ≥ 0 .
- b. i must be ≥ 1 .
- c. The value of $i+j-1$ must be $\leq k$.

Thus, the substring as specified by i and j must lie within s for the value of SUBSTR to be defined. If j is not specified, it is assumed to be equal to the value of $k-i+1$ i.e., it is assumed to be the remainder of the string starting at the ith position. If j is zero, the result is the null string.

31 MARCH 1972

MISCELLANEOUS FUNCTIONSDATE

The DATE function returns the current date as a character string of length 7, yymmdd where:

yy is the current year.

mm is the first three letters of the month.

dd is the current day.

Any use of the DATE function must be within the scope of a declaration of the identifier DATE with the BUILTIN attribute.

INLINE(f, r, s, t), INLINE(f, q, x, a, y, b, z, c)

The INLINE procedure is used to insert arbitrary STAR machine-instructions inline at compile time. Each parameter specifies a byte of the instruction in left to right order. The first operand, f, specifies the function code of the STAR instruction. The 32-bit instructions require four operands and the 64-bit instructions require eight operands. The operands r, s, and t specify the numbers of the STAR registers; q specifies the 8-bit sub-function designator; a, b, and c specify string or vector descriptor registers; x, y, and z specify index or offset registers. The f and q operands must be numeric constants; the rest of the operands may be numeric constants, variables, or arithmetic expressions.

If a variable of REGISTER storage-class is used as an operand, the number of the register allocated to that variable is inserted in the instruction; otherwise, for variables not stored in a register, the number of the register containing a descriptor of the variable is inserted in the instruction.

Example:

```
DECLABE SOURCE CHAR(80),
        TARGET CHAR(100);
```

```
CALL INLINE("F8", 5, 0, SOURCE, 0, "20", 0, TARGET);
```

In this example the registers containing the descriptors of SOURCE and TARGET are used by the "move characters" instruction to move the 80-character source string into the target string and fill the remaining 20 characters with blanks.

Appendix 1 -- Built-in Functions, Procedures,

31 MARCH 1972

When an arithmetic expression is used as an operand, the expression is first evaluated so all of the instructions necessary to evaluate the expression will precede the instruction being produced by `INLINE`. The number of the register containing the arithmetic result is then inserted into the instruction produced by `INLINE`.

Example:

```

DECLARE J          FIXED BINARY(47),
        COUNTER    FIXED REGISTER,
        KEYWORD    FLOAT REGISTER,
        TABLE(500) FLOAT DECIMAL(14) AUTO,
        MASK(5)    FLOAT DECIMAL(14)
                    CONSTANT("FFFF00000000FFFF",
                              "FFFF000000FFFFFF",
                              "FFFF0000FFFFFFFF",
                              "FFFF00FFFFFFFF",
                              "FFFFFFFFFFFFFFFF");

COUNTER = 0;

CALL INLINE("FF", 0, COUNTER, DESCR( 100, TABLE),
           0, DESCR( 1, KEYWORD),
           J+2, MASK);

```

This example will cause the compiler to generate code for constructing the descriptors of `TABLE` and `KEYWORD` and evaluating the expression `J + 2` before it generates the "search for masked key word" instruction. The registers containing the results of the `DESCR` function are used in the "FF" instruction which searches the first 100 elements of `TABLE` for a match with the contents of `KEYWORD` masked by `MASK(J+2)`.

If `INLINE` is used as a function, its value is the contents of the right-most register `t` or `c`. Thus one can write:

```

DCL R          FIXED REGISTER,
    COUNT(10)  FIXED AUTOMATIC,
    PATTERN    BIT(100);

COUNT(5) = INLINE("1F", PATTERN, 0, R);

```

which would cause the compiler to emit an instruction to count the number of one bits in the `PATTERN` followed by an instruction to store the result in `COUNT(5)`.

31 MARCH 1972

TIME

The TIME function returns the current time of day as a character string of length 11, hh:mm:ss.dd where:

- hh is the current hour of the day
- mm is the current minute within the hour
- ss is the current second within the minute
- dd is the decimal fraction of the current second.

Any use of the TIME function must be within the scope of a declaration of the identifier TIME with the attribute BUILTIN.

31 MARCH 1972

APPENDIX 2 - CONDITIONSINTRODUCTION

For each condition, the description in this appendix includes the circumstances under which the condition is raised, the standard system action that would be taken in the absence of programmer-specified action, and, where applicable, the result.

Conditions may be specified in the ON, REVERT, and SIGNAL statements (see Chapter 8 - Statements and Chapter 7 - Interrupt Handling).

If no ON statement is currently in effect when a condition is raised, the standard system action for that condition is taken.

CONVERSION CONDITION

The CONVERSION condition can be raised whenever an illegal conversion is attempted within the conversion built-in functions: FIXED, FLOAT, CHAR, BIT, and ENTRY, or execution of a GET statement. Conversion across the equal sign in an assignment statement (see Chapter 4 - Data Manipulation), and implicit fixed to float and float to fixed conversions will not raise the CONVERSION condition.

Result: The result is undefined. On normal return from the on-unit, the ERROR condition is raised.

Standard system action: Comment and raise the ERROR condition.

31 MARCH 1972

ENDFILE CONDITION

The condition of the form: ENDFILE (file-variable) may be raised during any GET operation on the Apple file referred to by the file-variable. It is caused by an attempt to read past the file delimiter. If the file is not closed after the ENDFILE condition is raised, any subsequent GET operations on the same file will raise the condition again. The execution of a SIGNAL ENDFILE (file-variable) statement will also raise the ENDFILE condition.

Result: On normal return from the on-unit, execution continues with the statement immediately following the statement which raised the ENDFILE condition.

Standard system action: Comment and raise the ERROR condition.

ERROR CONDITION

The ERROR condition is raised by: (1) the standard system action taken when another condition is raised which includes the raising of the ERROR condition, (2) the result of an error, for which there is no condition, occurring during program execution, and (3) the execution of a SIGNAL ERROR statement.

Result: On normal return from the on-unit, the FINISH condition is raised.

Standard system action: Comment and raise the FINISH condition.

FIND CONDITION

The FIND condition is raised whenever (1) the FIND statement with no ELSE clause is executed, and the specified entity cannot be found, and (2) the INSERT statement with the BEFORE or AFTER option is executed and the specified entity cannot be found.

Standard system action: Comment and raise the ERROR condition.

31 MARCH 1972

FINISH CONDITION

The FINISH condition is raised by: (1) the standard system action taken for the ERROR condition, (2) the action taken on normal return from the on-unit for the ERROR condition, (3) the execution of a statement that would cause termination of an Apple program (an EXIT statement), and (4) the execution of a SIGNAL FINISH statement.

Result: On normal return from the on-unit, the program is terminated.

Standard system action: The program is terminated.

OVERFLOW CONDITION

The OVERFLOW condition is raised when the exponent of a floating-point number exceeds the permitted maximum. This maximum is 8630 for long float, and 33 for short float.

Result: On normal return from the on-unit, program execution continues near the point of overflow. The value of the floating-point number is set to an undefined value.

Standard system action: Comment and raise the ERROR condition.

PROGRAMMER-DEFINED CONDITION

The condition of the form: CONDITION (identifier) allows a programmer to establish an on-unit that will be executed whenever a SIGNAL statement is executed specifying CONDITION and the identifier. Programmer-defined conditions must be declared with the CONDITION attribute. The programmer-defined condition can only be raised by a SIGNAL statement specifying that condition.

Standard system action: Comment and raise the ERROR condition.

31 MARCH 1972

STORAGE CONDITION

The STORAGE condition is raised by: (1) an attempt to allocate a based variable in an Apple file that contains insufficient free storage for the allocation to be made, or (2) the execution of a SIGNAL STORAGE statement.

Result: If the condition is raised due to insufficient free storage being available for an allocation to be made, on normal return from the on-unit, the options in the ALLOCATE statement are reevaluated and the allocation is attempted again. If the condition was raised by a SIGNAL statement, on normal return from the on-unit, the statement following the SIGNAL statement is executed.

Standard system action: Comment and raise the ERROR condition.

UNDEFINEDFILE CONDITION

This condition is raised by the PUT, GET, ALLOCATE, FOR EACH, INSERT, REMOVE, FIND, LET, and FREE statements, and the use of the CHAR built-in function to convert from a file value to a character string, if the file referenced in these statements has not been opened.

Result: On normal return from an on-unit, execution continues with the next statement.

Standard system action: Comment and raise the ERROR condition.

UNDERFLOW CONDITION

The UNDERFLOW condition is raised when the exponent of a floating-point number becomes smaller than the permitted minimum. This minimum is -8630 for float long, and -33 for float short.

31 MARCH 1972

Result: On normal return from the on-unit, program execution continues near the point of underflow. The value of the floating-point number is set to zero.

Standard System action: Comment and continue.

ZERODIVIDE CONDITION

The ZERODIVIDE condition is raised when an attempt is made to divide by zero. This occurs if the divisor is zero.

Result: On normal return from the on-unit, execution continues near the point of zero-divide. The quotient is set to indefinite.

Standard system action: Comment and raise the ERROR condition.

31 MARCH 1972

APPENDIX 3 - KEYWORDS, ABBREVIATIONS AND SYNONYMSKEYWORD ABBREVIATIONS

Abbreviations are provided for certain keywords. The abbreviations themselves are keywords and will be recognized as synonymous in every respect with the unabbreviated keywords.

<u>KEYWORD</u>	<u>ABBREVIATION</u>
A	
AFTER	
ALIGN	
ALLOCATE	
ANY	
AUTOMATIC	AUTO
B	
BASED	
BEFORE	
BEGIN	
BINARY	BIN
BIT	
BUILTIN	
BY	
CALL	
CASE	
CHARACTER	CHAR
COLUMN	COL
CONDITION	COND
CONSTANT	
CONTAINING	
CONVERSION	
DECIMAL	DEC
DECLARE	DCL
DO	
E	
EACH	
EDIT	
END	
ENDFILE	EOF
ENTITY	
ENTRY	
ERROR	

31 MARCH 1972

EVENT	
EXIT	
EXTERNAL	EXT
F	
FILE	
FIND	
FINISH	
FIRST	
FIXED	
FLOAT	
FOR	
FREE	
FROM	
GET	
GO	
GOTO	
IF	
IN	
INCLUSIVE	
INITIAL	INIT
INSERT	
INTERNAL	INT
LABEL	
LAST	
LET	
LIKE	
LINE	
LOCK	
NEAR	
OF	
OFFSET	
ON	
OVERFLOW	OFL
PAGE	
POINTER	PTR
PROCEDURE	PROC
PUT	
REGISTER	REG
REMOVE	
REMOTE	
RETURN	
RETURNS	
REVERT	
SET	
SIGNAL	
SKIP	
STATIC	
STRING	
STORAGE	
THEN	

31 MARCH 1972

TO	
UNDEFINEDFILE	UNDF
UNDERFLOW	UFL
UNLOCK	
UNTIL	
VARIABLE	VAR
VARYING	
WAIT	
WHILE	
WITH	
X	
ZERODIVIDE	ZDIV

31 MARCH 1972

KEYWORD SYNONYMS

The following list of synonyms is provided for those programmers who have written PL/I programs using the APL statements. Each synonym will be recognized by the Apple compiler as the indicated language keyword.

<u>SYNONYM</u>	<u>APPLE KEYWORD</u>
CREATE	ALLOCATE
CALLED	SET
DELETE	FREE
ENTITY_VARIABLE	POINTER
E_VAR	PTR
ENTITY_SYSTEM } E_SYS }	ENTITY
ENTITY_SET } E_SET }	FILE_SET
SET_VARIABLE	POINTER
S_VAR	PTR
IN	ON

APPENDIX 4 - DATA CHARACTER SET

The Apple data character set is the ASCII character set defined in TABLE 1. TABLE 2 describes the control character subset and TABLE 3 the graphic character subset, excluding A-Z, a-z, and 0-9.

					b8	0	0	0	0	0	0	0	0	0
					b7	0	0	0	0	1	1	1	1	1
					b6	0	0	1	1	0	0	1	1	1
					b5	0	1	0	1	0	1	0	1	1
					COL	0	1	2	3	4	5	6	7	
b4	b3	b2	b1	ROW										
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p		
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q		
0	0	1	0	2	STX	DC2	"	2	B	R	b	r		
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s		
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t		
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u		
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v		
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w		
1	0	0	0	8	BS	CAN	(8	H	X	h	x		
1	0	0	1	9	HT	EM)	9	I	Y	i	y		
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z		
1	0	1	1	B	VT	ESC	+	;	K	[k	{		
1	1	0	0	C	FF	FS	,	<	L	\	l			
1	1	0	1	D	CR	GS	-	=	M]	m	}		
1	1	1	0	E	SO	RS	.	>	N	~	n	~		
1	1	1	1	F	SI	US	/	?	O	_	o	DEL		

TABLE 1 - APPLE DATA CHARACTER SET

The remaining internal codes (b8=1) are undefined.

31 MARCH 1972

NUL Null	DLE Data Link Escape (CC)
SOH Start of Heading (CC)	DC1 Device Control 1
STX Start of Text (CC)	DC2 Device Control 2
ETX End of Text (CC)	DC3 Device Control 3
EOT End of Transmission (CC)	DC4 Device Control 4
ENQ Enquiry (CC)	NAK Negative Acknowledge (CC)
ACK Acknowledge (CC)	SYN Synchronous Idle (CC)
BEL Bell (audible or attention signal)	ETB End Transmission Block (CC)
BS Backspace (FE)	CAN Cancel
HT Horizontal Tabulation (punched card skip) (FE)	EM End of Medium
LF Line Feed (FD)	SUB Substitute
VT Vertical Tabulation	ESC Escape
FF Form Feed (FE)	FS File Separator (IS)
CR Carriage Return (FE)	GS Group Separator (IS)
SO Shift Out	RS Record Separator (IS)
SI Shift In	US Unit Separator (IS)
	DEL Delete

Note: (CC) Communication Control
(FE) Format Effector
(IS) Information Separator

TABLE 2 -- CONTROL CHARACTERS

31 MARCH 1972

<u>COLUMN/ROW</u>	<u>SYMBOL</u>	<u>NAME</u>
2/0		BLANK
2/1	!	OR SYMBOL
2/2	"	DOUBLE QUOTATION MARK
2/3	#	NUMBFR SIGN
2/4	\$	DOLLAR SIGN
2/5	%	PERCENT SIGN
2/6	&	AND SYMBOL
2/7	'	SINGLE QUOTATION MARK
2/8	(LEFT PARENTHESIS
2/9)	RIGHT PARENTHESIS
2/A	*	ASTERISK OR MULTIPLY SYMBOL
2/B	+	PLUS
2/C	,	COMMA
2/D	-	MINUS
2/E	.	PERIOD OR DECIMAL POINT
2/F	/	SLASH OR DIVIDE SYMBOL
3/A	:	COLON
3/B	;	SEMICOLON
3/C	<	LESS THAN
3/D	=	EQUAL OR ASSIGNMENT SYMBOL
3/E	>	GREATER THAN
3/F	?	QUESTION MARK
4/0	@	AT SYMBOL
5/B	[LEFT BRACKET
5/C	\	REVERSE SLASH
5/D]	RIGHT BRACKET
5/E	~	NOT SYMBOL
5/F	␣	BREAK CHARACTER
6/0	`	GRAVE ACCENT
7/B	{	LEFT BRACE
7/C		VERTICAL LINE
7/D	}	RIGHT BRACE
7/E	~	TILDE

TABLE 3 - GRAPHIC CHARACTERS

31 MARCH 1972

APPENDIX 5 - COMPILE-TIME CONTROLS

General Format:

```
control-statement ::= % command[ ( parameter-list ) ];
```

General Rules:

1. A control statement may appear anywhere that a statement may appear in an Apple source program.
2. The commands which control the printed listing, such as % PAGE; % SKIP (3); or % LIST (CODE); are performed at the point where they occur in the printed listing. Other commands merely request that some information be printed at the end of the printed listing such as % LIST(OBJECT);.
3. The commands which set a switch for a continuing action, (such as % LIST (CODE); which causes the emitted object code to be listed) may be turned off by a counter-command that is formed by prefixing the letters "NO" to the command, such as % NOLIST (CODE);.

31 MARCH 1972

The following table lists the currently defined commands and their function.

<u>COMMAND</u>	<u>FUNCTION</u>
% SKIP(n)	Skip n lines on the printed listing. The parameter n must be an unsigned integer greater than zero.
% PAGE;	Eject to the top of a new page on the listing.
% LIST(option,...)	Insert into the printed listing additional information designated by the options. where the options may be:
CODE	List the emitted instructions
MACRO	List the expansions made during the literally expansions (see Appendix 8).
WARNINGS	List the warning messages generated during compilation
OBJECT	Dump the object module in hexadecimal
CONTROLS	List the compile-time page controls (% PAGE and % SKIP) where they occur in the source text.
% INCLUDE file-name[(entity)]	Replace this command with the source text contained in the named file, or entity in the named file.
% DECLARE	See Appendix 8.

31 MARCH 1972

APPENDIX 6 - NOTATION

The entire manual uses a uniform system of notation that is not a part of the language. This notation describes the syntax or construction of the language. The following rules describe the use of this notation:

1. A notation variable consists of lower-case letters and hyphens, must begin with a letter, and may be enclosed in braces.

Examples:

- | | | |
|----|--------------|---|
| a. | digit | this denotes the occurrence of a digit 0 through 9 inclusive. |
| b. | do-statement | this denotes the occurrence of a DO-statement. |

2. A notation constant appears in upper-case and denotes the literal occurrence of the indicated characters. The constant may be enclosed in braces and is defined by the syntax of the language.

Example:

DECLARE	this denotes the occurrence of the keyword DECLARE.
---------	---

3. The term "syntactic unit" is used in the following as:
 - a. a single variable or constant
 - b. a collection of notation variables, notation constants, and symbols enclosed in braces or brackets
4. Braces with the vertical stroke (|) or vertical stacking of syntactic units indicates that a choice is to be made.

31 MARCH 1972

Examples:

```

identifier      { FIXED }
                 { FLOAT }

identifier {FIXED|FLOAT}

```

5. Anything enclosed in square brackets may appear once or not at all. Vertical stacking with square brackets indicates that no more than one of the stacked units can appear.

Example:

```
[ VARYING ]
```

6. The ellipsis (...) denotes the occurrence of the preceding unit one or more times.

Example:

```
digit ...
```

7. Most notation variables are defined in terms of a general format. A general format is a sequence of the name being defined, followed by the definition symbol (::=), followed by the definition. The definition symbol says that the named item "can be represented by" the indicated definition. The definition may involve notation constants or other named notation variables.

Example:

```
scope-attribute ::= EXTERNAL|INTERNAL
```

8. Blanks appearing in formats do not represent the character blank of the language character set. Blanks are used as delimiters of the syntax and to improve the readability of the definitions. Any two notation variables or constants are separated by blanks. Any explicit use of the character blank of the language character set will be denoted by a b.

31 MARCH 1972

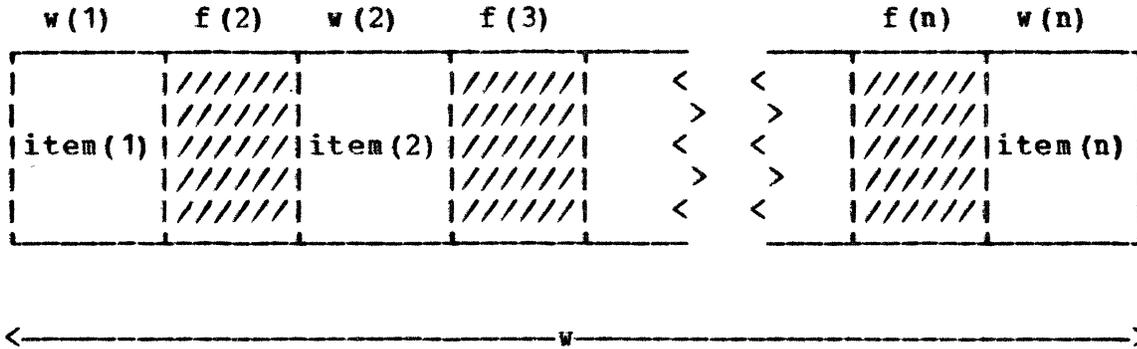
APPENDIX 7 - STRUCTURE MAPPING

The problem of mapping a data structure concerns the assignment of addresses to the elements of a structure and at the same time ensuring that the addresses meet the hardware requirements for access. In Apple, all data will be stored so that it may be accessed as rapidly as possible, even at the expense of some wasted space. To do this, the hardware requires that bit addresses be some multiple of an alignment factor that is defined for each data type. The alignment factors are as follows:

<u>Data type</u>	<u>Alignment Factor</u>	<u>Length</u>
BIT (n)	1	n
CHARACTER (n)	8	8n
VARYING	8	8 (n+1)
ENTRY	64	192
EVENT	64	64
FILE	64	64
FIXED or FLOAT BINARY (p)		
1 ≤ p ≤ 23	32	32
24 ≤ p ≤ 42	64	64
LABEL	64	128
OFFSET	32	32
POINTER	64	64
DESCRIPTOR	64	64

31 MARCH 1972

The layout of a structure in storage has the following form:



where the length of the i -th item is $w(i)$ and its alignment factor is $a(i)$. The fields of length $f(2), f(3), \dots, f(n)$ are padding introduced so that $item(2), \dots, item(n)$ will have the correct alignment. The total length of the mapped structure is $w = w(1) + f(2) + w(2) + \dots + f(n) + w(n)$ and the alignment factor is $a = \text{MAX}(a(1), \dots, a(n))$. The following algorithm defines the method by which the lengths of the padding are calculated. The algorithm assumes that it is mapping a set of elements whose alignment and length are determined. If these elements are substructures they must have been mapped using the same algorithm.

```

a = MAX(a(1), ..., a(n));
w = w(1);
DO k = 2 TO n;
    f(k) = a(k) * FLOOR((w+a(k)-1)/a(k)) - w;
    w = w + f(k) + w(k);
END;
```

31 MARCH 1972

APPENDIX 8 -- LITERALLY

The LITERALLY compile-time specification allows the programmer to designate that certain identifiers will be replaced by specified character strings throughout the external procedure before compilation.

General format:

literally-specification ::=

% DECLARE substitution-specification ;

substitution-specification ::=

identifier [LITERALLY] [(parameter-list)]
character-string-constant

parameter-list ::= parameter [, parameter] ...

General rules:

1. The scope of the literally-specification is the whole of the external procedure and all its contained blocks.
2. The specification states that the compiler should replace each appearance of the identifier within the scope of the specification by a character string before compilation. The replacing string is then to be scanned for further replacements.
3. If no parameter list follows the keyword LITERALLY, then the identifier is to be replaced by the "character-string-constant".
4. If the keyword LITERALLY is followed by a parameter list, then each subsequent occurrence of the identifier must be followed by an argument list with the same number of arguments as there are

31 MARCH 1972

parameters. The character string that is used to replace the identifier is then constructed by replacing in the "character-string-constant", each occurrence of the characters that form the parameter identifiers by the characters in the corresponding argument position.

5. The replacing character string must not be used to form a part of a syntactic unit except for strings.
6. A literally-specification may not be given for any keyword listed in Appendix 3.
7. The literally-specification must occur before the first appearance of the identifier.
8. An argument must not contain unbalanced parentheses.
9. An argument must not contain the character ',' unless it is contained within parentheses, possibly with other characters. The parentheses will then form part of the replacing character string.

Examples:

1. Literally specification without parameters:

```
% DECLARE SYTYPE LITERALLY 'SYT(12)';
```

All appearances of the identifier SYTYPE within the scope of the above specification will be replaced by SYT(12), thus the statement:

```
IF SYTYPE = 235 THEN ...
```

will be compiled as though

31 MARCH 1972

```
IF SYT(12) = 235 THEN ...
```

had been written.

2. Literally specification with parameters:

```
% DECLARE BITS LITERALLY(A1, A2) 'A1 * A2 * 64';
```

Within the scope of this specification, the statement

```
I = BITS(J, 8);
```

will be compiled as

```
I = J * 8 * 64;
```

and, assuming the specification in Example 1 above,

```
I = BITS(SYTYPE, A(I, J));
```

will be compiled as:

```
I = SYT(12) * A(I, J) * 64;
```

3. % DECLARE STRUCT(NAME)
 '1 NAME BASED,
 2 N FIXED,
 2 X(2*I REFER (NAME.N)) FLOAT';
 DECLARE STRUCT(A),
 STRUCT(B);

The above declarations are equivalent to the following:

```
DECLARE 1 A BASED,  
      2 N FIXED,  
      2 X(2*I REFER (A.N)) FLOAT,  
      1 B BASED,  
      2 N FIXED,  
      2 X(2*I REFER (B.N)) FLOAT;
```

31 MARCH 1972