SIMPLE LANGUAGE SPECIFICATION

REFERENCE MANUAL

Mark Greenberg

July 26, 1973

Revised September 5, 1973

Center for Research in Management Science
Systems Group

Technical Document

R-1

0.5 compiler differences from 1.0

1) may not re-use name declared as scalar constant
2) may not use STRING in formal argument list
3) DESCRIPTOR statement not implemented
4) array fail actions not implemented
5) NOT is same precedence as GOTO
6) FRETURN values not implemented
7) ANY type not implemented
8) function calls precedence is just lower than indirection
9) ENDWHILE not implemented

TABLE OF CONTENTS

1. Introduction.

This document provides a complete description of the SIMPLE language.

SIMPLE is a programming language system especially designed to facilitate the development of large programming systems such as operating systems and language systems. SIMPLE is not intended to be a general purpose language; however, its useful application may very well exceed the scope of system programming.

The SIMPLE language was designed by the author with the help of Charles Grant, Paul McJones and David Redell. Its design was influenced by the QSPL language developed for the Berkeley SDS-940 Time-Sharing System.

The SIMPLE language includes the following features:

(1) a complete block structured control mechanism for specifying iterative and conditional control structures;

(2) convenient methods for accessing and manipulating partial word values, one dimensional arrays of various element sizes, strings of characters, and ring buffers;

(3) a character and number oriented input-output facility;

(4) recursive functions;

(5) a token substitution macro-facility.

The SIMPLE language system is a totally integrated, interactive system that includes a program text editor, a compiler, and a source language level debugger (see SIMPLE Interactive Language System Reference Manual). Code is compiled for an instruction processing unit that was designed in parallel with the SIMPLE language and implemented as a microprogram (called the SIPU) for the META4 microprocessor (see SIMPLE Instruction Processing Unit Reference Manual). A collection of standard functions called the SIMPLE Runtime is provided as part of the integrated language system (see SIMPLE Runtime Reference Manual).

2. SIMPLE Character Set.

The SIMPLE language character set consists of 65 characters: the 26 letters A-Z, ten digits Ø-9, the 27 special characters:

! " # % & ' ( ) * + , - . /

$: ; < = > ? @ [\ ] ↑ ←

blank and carriage return.

All other characters which may exist in a SIMPLE program are ignored by the SIMPLE compiler in determining the content of a SIMPLE program.

The character codes for the characters are given in Appendix A.

3. SIMPLE Program Lexical Structure.

A SIMPLE program consists of a number of blocks. All blocks are function blocks which specify a function body except for the first block in the program which is the global declaration block. Each block is made up of statements and comments.

3.1 Comments

A comment begins (1) with an asterisk which is the first non-blank character following a statement, another comment, or the beginning of the block, or (2) with a double dot (..) anywhere within a program except inside a quoted string. A comment is terminated with a carriage return. The string of characters in a comment is completely ignored by the compiler except that a comment within a statement serves to delimit primaries.

3.2 Statements

A statement is a string of characters which begins following another statement, a comment or the beginning of the block and ends with a carriage return or a semi-colon not enclosed in quotes. A carriage return which terminates a double dot comment does not end the statement also. Thus, statement continuation to the next line can be achieved by ending a line with a double dot.

The following example contains five comments and two statements.

Example:

```
* THIS IS A COMMENT
      A ← B+C; * ANOTHER COMMENT
      F( A,    .. COMMENT FOR A
         B,    .. COMMENT FOR B
         C)    ;* COMMENT FOR C
```

3.3 Primaries

The string of characters in a statement is divided into a string of primaries which may be of several types:

(1) names

(2) numbers

(3) character constants

(4) string constants

(5) operators.

a) A name is any number of alphanumeric or blackslash characters beginning with a letter or blackslash (only the first 127 characters are relevant, the rest are ignored). For programs compiled in runtime mode, percent sign ('%') is also a legal character for a name.

b) A number is any number of digits optionally followed by a 'B' optionally followed by another string of digits. The 'B' indicates the number is octal, otherwise it is decimal. The optional string of digits is interpreted as a decimal number which is a scaling factor indicating the number of octal zeros to append to the number. All constants are truncated to 32 bits. The numbers in the following example all have the same value.

Example:

```
400B
4B2
256
```

c) A character constant is any one to four pseudo-characters enclosed in single quotes. The value of the character constant is the values of the pseudo-characters right justified in a 32-bit word, zeros filling in unspecified characters.

d) A string constant is any number of pseudo characters enclosed in double quotes.

A pseudo character is:

(1) any character other than ampersand, carriage return, single quote or double quote;

(2) An ampersand followed by one to three octal digits. The octal number truncated to 8 bits defines the value of the pseudo character. (Example: &155 is a carriage return character.)

(3) An ampersand followed by a letter. The value of the pseudo character is the code for the letter plus 100B . (Example: &A is equivalent to &141.) This allows easy specification of the control characters.

(4) An ampersand followed by any other character. The ampersand serves as an escape and the following character is taken literally. (Example: && is a single ampersand.) This allows single quotes and double quotes to be inserted in character and string constants, respectively.)

Carriage returns are illegal within character or string constants thus preventing runaway compilations when a programmer neglects to insert a closing quote. Within string constants only, an &← will cause all subsequent characters up to and including a carriage return to be ignored. This provides a method for line continuation within string constants.

e) An operator is any special character other than single quote or double quote.

4. Statement Syntax Notation.

The syntax of SIMPLE statements will be specified in an extended BNF-like notation which is now described.

(1) Lower case names are syntactic types. Syntactic type names used in descriptive text are enclosed in angle brackets for clarity. When the same syntactic type is used more than once in a BNF statement a number may be appended to the syntactic type so that unambiguous references to the syntactic types may be made in the descriptive text.

(2) Upper case names are literals (such as reserved words).

(3) Characters enclosed in single quotes are literals.

(4) The construct [x] indicates that x is optional.

(5) The construct (x) indicates that exactly one occurrence of x must be present. Parentheses are used for grouping.

(6) The construct arb(x) indicates that zero or more occurrences of x may be present.

(7) The construct sep(x, y) indicates that one or more occurrences of x must be present separated by occurrences of y .

(8) Successive constructs indicate concatenation.

(9) / is used to indicate alternation.

5. Memory.

All data which may be referenced from expressions of a SIMPLE program is stored in the "memory" of the SIMPLE program. Memory is divided into two parts, variable memory and constant memory. Variable memory is subdivided into three kinds:

(1) global memory

(2) local memory

(3) external memory.

Constant memory is read-only. It holds the constant space for vectors initialized at compile time plus the values of all constants used in expressions. Global memory is allocated at compile time and never becomes de-allocated. Space for global variables, global vectors, and local vectors are allocated in global memory. Local memory is allocated on function call and deallocated on function return. Local memory is used for allocation of local variables. External memory is supplied through the capability machinery by the operating system. Data in external memory may be accessed with "memory reference expressions" in exactly the same way that global memory is referenced.

6. Values.

All values manipulated by expressions in SIMPLE programs are of two kinds:

6.1 Scalars

A scalar is a one word (32-bit) value. It is the fundamental unit of information manipulated by expressions of a SIMPLE program. Scalars may have a number of specialized interpretations when used as the operands of certain operators.

(1) a scalar used as the right operand of a field tailing operator is interpreted as a field selector (see SIPU manual for details).

(2) a scalar used inside square brackets of a subscripting operation is interpreted as a positive integer index.

(3) a scalar used as the operand of a GOTO operator is interpreted as an instruction address relative to the beginning of the code for the currently executing function.

(4) a scalar used as the second operand of a ring access operator is interpreted as a ring selector (see SIPU manual for details).

(5) a scalar used preceding parenthesis in a function call operation is interpreted as a function identification number (see SIPU manual for details).

Details of the use of scalars are given with the semantic descriptions of the relevant operators in Section 8.2.

6.2 Descriptors

A descriptor is a two word value. There are two kinds of descriptors.

(1) Vector descriptors provide a complete description of a "vector" in memory. A vector is a linear, contiguous, ordered set of fixed sized values or "elements" stored in memory. A vector descriptor specifies

a) The location of the vector in global, constant, or external memory.

b) The number of elements in the vector which must be between 1 and 64K.

c) The element size of the vector which may be 8 bits, 16 bits, or 1 to 127 words.

d) An index origin of one or zero.

e) A two's complement sign extension option for 8 bit, and 16 bit element sizes (see SIPU manual for details).

(2) References. A reference specifies the location in memory and the size of a value stored in memory. The value specified may be any continuous sub-set of the bits of a single word or any 1 to 127 words (see the SIPU manual for details).

7. Names.

Names are used in SIMPLE to represent various kinds of objects. The meaning of a name is determined by its declaration. Every name must be declared. A name may be declared

(1) in a declaration statement,

(2) when used as a label beginning a statement,

(3) when used as the control variable of an iteration control block, or

(4) in a FUNCTION statement.

7.1 Scope of names

Every name has a scope which limits the context in which the declared meaning of a name is valid. A name may have global or local scope. A name is global if it is declared in the global declaration block or if it is the name declared as a function in a FUNCTION statement. Also, there are a number of reserved words (see Appendix D) which are pre-declared and have global scope. Furthermore, the global names of the runtime are also global to the user program. All other declarations are within some function block. The scope of such names is local and their declared meaning is valid only in the function block in which they are declared.

A particular name may be declared in any number of blocks. If declared in several function blocks, the name will have a different meaning in each function block. If a name is declared in both a function block and the global declaration block, then within the function block the locally declared meaning prevails. Names declared globally as macros cannot be redeclared locally. It is a program error to redeclare a reserved word name. It is also a program error if a name used within a function block is not declared globally or declared within the function block.

In general, a name declared within a block may not be redeclared within the same block. However, names declared as scalar constants may be redeclared an arbitrary number of times as scalar constants with different values using the CONSTANT declaration statement. The value used in a particular reference to the name will be that specified in the textually preceding declaration of the name.

7.2 Interpretation of names

Names may be declared to represent one of several kinds of objects

(1) operators

(2) macros

(3) values in memory.

All operators are pre-declared reserved words and are used in the same contexts as special characters. (Examples: AND, IF, FUNCTION.) See Appendix D for complete list.

Macros provide a method by which use of a name declared as a macro can be viewed as equivalent to an arbitrary string of primaries. Macros are discussed in greater detail elsewhere.

Memory value names provide a symbolic way of referring to values in memory. These names can refer to values in constant memory or variable memory and are said to be declared as constants or variables, respectively. Names declared as constants must have initial values specified in their declarations whereas names declared as variables may not.

7.3 Types for names

Names for values in memory have a type associated with them; scalar, vector, reference, descriptor or any. This type is specified when the name is declared. Names declared as type vector, reference, or descriptor can also have associated with them one additional level of type specification which indicates the type of the elements in memory to which the descriptor refers. Scalar names declared in FIELD statements can also have a type specification for the field referenced by the field selector. The type of an element in memory is specified with a <type> construct which is interpreted as follows:

word	type
SCALAR	scalar
VECTOR	vector
RING	vector
REFERENCE	reference
STRING	reference
DESCRIPTOR	descriptor
ANY	any

8. Expressions

Syntax:

```
expression      = sep(where, '&')
where           = conditional [WHERE conditional]
conditional     = IF or (DO / THEN) or [ELSE conditional] / or
or              = sep(and, OR)
and             = sep(not, AND)
not             = [NOT] relational
relational      = rem [('=' / '#' / '>' / '>=' / '<' / '<=') rem]
rem             = sep(addition, REM)
addition        = sep(multiplication, ('+' / '-'))
multiplication  = sep (unary, ('*' / '/' / CYCLE / SHIFT))
unary           = return / exit / sign
return          = (RETURN / SRETURN / FRETURN) [conditional]
exit            = EXIT [name]
sign            = ['+' / '-' / GOTO] assignment
assignment      = indirection [ '←' conditional]
indirection     = arb('$' / '@') tailing
tailing         = sep(bottom, ('$' / '.' / '@'))
bottom          = logical / absolute / code / ringaccess / functioncall /
                  subscripting
logical         = (AND / OR / EOR) '(' expression ',' expression')'
absolute        = ABS '(' expression ')'
code            = CODE '(' initlist ')'
```

```
ringaccess      = (FRONT / REAR / GETF / GETR / PUTF / PUTR)
                  '(' expression [',' expression] [failaction] ')'
functioncall    = primary '(' [actuals] [failure] ')'
actuals         = sep(expression, ',')
failure         = ':' [ '[' name ']' ] [expression1]
subscripting    = primary arb('[' expression [failaction] ']')
primary         = '(' expression ')' / constant / variable
failaction      = ':' expression
```

8.1 General Semantics

8.1.1 Definition.

The term "expression" is used in this manual to mean any sequence of operands and operators which conform to the stated syntax and semantic rules for expressions. Thus, any operand of some operator is itself an expression and a sub-expression of a larger expression.

8.1.2 Syntax.

Two types of syntactic constructs are used to associate an operator with its operand(s): infix syntax and function syntax. For example:

X + Y	infix syntax
AND(X, Y)	function syntax

In particular, the bitwise logical and ring access operators use function syntax while most others use infix syntax. In addition, the function call and square brackets, respectively, immediately following a <primary>. For example:

F(A)	function call
V[I]	vector subscripting

8.1.3 Simple and complex expressions.

Every expression is either "simple" meaning it has no operator or else it is complex and has a "principle operator" which is the operator of lowest precedence in the outermost level of parenthesis or square bracket nesting. For example, the '*' is the principle operator in the following expression:

```
(A + B) * V[I]
```

If an expression is simple then it consists of a single variable name or a single constant, optionally enclosed by any number of balanced parentheses.

8.1.4 Associativity.

Operators of the same precedence are left associative. For example, the following two expressions are equivalent:

```
A + B - C
(A + B) - C
```

However, the assignment operator ('←') is right associative.

8.1.5 Order of evaluation.

Every complex expression evaluates its principle operator last after evaluating the operands of the principle operator. This rule recursively applied to the operands of the principle operator as sub-expressions specifies the order of evaluation of an expression.

Note that for most binary operators (exceptions specified in operator semantics) the order in which the operands are evaluated is unspecified. Note also that certain operators (e.g., boolean AND) supress evaluation of one operand conditionally on the evaluation of the other.

8.1.6 Memory reference expressions.

A "memory reference expression" is an expression which refers to a value in the memory of a SIMPLE program. A memory reference expression is either a simple variable or a complex expression which satisfies one of the following conditions:

a. principle operator is '.' field tailing,

b. principle operator is '$' field tailing, and left operand of '$' is a memory reference expression,

c. principle operator is vector subscripting ([]) ,

d. principle operator is indirection ('$') ,

e. principle operator is a ring access operator (FRONT, REAR, GETF, GETR, PUTF, PUTR)

8.1.7 Constant expressions.

A "constant expression" is either a simple constant or a complex expression satisfying both the following rules:

a. all operands of the principle operator are themselves constant expressions,

b. the principle operator can be evaluated as a constant according to the rules in Appendix E.

8.1.8 Result of evaluation.

Every expression when evaluated yields a "result." A result has a number of attributes:

a. a value,

b. a value size,

c. a type, and

d. a reference (for memory reference expressions only).

Some operators (e.g., FOR, GOTO) yield unspecified results when evaluated. Memory reference expressions include a reference as part of the result which is used instead of the value in certain contexts such as the left operand of an assignment operator.

Result type.

Evaluation of an expression yields a result with one of the following types:

a. scalar,

b. vector,

c. reference.

In addition, a result may be of one of two composite types. A result is of type "descriptor" if it is of type vector or of type reference, and thus, is legal in any context where an operand of type vector or reference is required and is interpreted semantically exactly as if it were of the required type. A result is of type "any" if it is of type scalar or vector or reference, and thus is legal in any context and is interpreted semantically exactly as if it is of the required type (exceptions of this rule are specifically noted for the assignment and function argument passing operations).

The specification of a type attribute of a result serves two distinct semantic purposes:

a. Type attributes are used to determine of one or two word values are to be copied by the assignment operator and during passing of function arguments. Details are discussed with the semantics of these operations.

b. Type attributes are used to test the validity of the operand(s) of an operator at compile time. The detailed rules of type validity checking for the operators is given in Appendix B.

The type machinery is not totally effective since types can be associated with values in memory to only one level of referencing and it is possible to construct expressions that have more than one level of referencing, e.g., A[I] [J]. Such expressions will evaluate to a result of type any and therefore improper uses of that expression may go unchecked.

Result value size.

A result value has a size which is specified as one or more words. The following rules must be observed for valid expression construction:

a. A result of size one word must be type scalar, or type any.

b. A result of type vector, reference or descriptor must be of size two words.

c. Sizes greater than two words can occur only as the result of memory reference expressions. Such results are referred to as "multi-words." A multi-word result can be legally used only as the left operand of a '$' field tailing operator or as the operand of an '@' reference operator. Only the reference of a multi-word result can be used, never the value.

If an element in memory, less than 32 bits in size, is evaluated with a memory reference expression then the result is extended to one word with the element value right justified. The part-word element value may be two's complement sign extended when evaluated if the sign extended option is specified in the vector descriptor, reference, or field selector.

8.1.9 Scalar Operators.

Most of the operators in SIMPLE expressions are "scalar operators." A scalar operator when evaluated generates a result of type scalar. All the operands of a scalar operator must be of type scalar. If the result type of an operand is type any then the result of that operand is used in evaluating the scalar operator as if it were of type scalar.

8.1.10 Evaluation of variables.

A simple expression which consists solely of a name declared as a variable is evaluated as follows. The result type is the type specified in the declaration of the name. The result value is the value of the variable in memory at the time of evaluation. The result size is two words for a variable of type vector, reference, or descriptor and one word otherwise. The result reference is to the variable in memory.

8.1.11 Evaluation of constants.

A constant in a simple expression may be one of the following:

a. A number, in which case the result is of type scalar, and size one word, with value specified by the number.

b. A string constant in which case the result is of type reference and size two words, the value being a reference to a three word string descriptor for a vector of 8-bit elements which is allocated and initialized with the characters of the string constant in constant memory.

c. A name declared to be a constant in which case the result value and type are those specified in the declaration of the name.

d. A character constant in which case the result is of type scalar with value specified by the character constant.

8.1.12 Semantics.

The semantics of all the operators are listed in Section 8.2 in order of precedence or binding strength, the lowest precedence (most loosely bound) operator first. Operators of the same precedence are listed together.

8.2 Semantics of Operators

& causes successive evaluation. The left operand is evaluated and then the right operand. The result is the result of the right operand.

Example: X ← Y & GOTO L

WHERE is similar to '&' but causes evaluation of the right operand before the left operand. The result is the result of the left operand. It may not be iterated.

Example: F(X, Y) WHERE X ← 1

IF
THEN DO
ELSE

specify conditional evaluation. These operators may appear in the general form

IF expr1 THEN expr2 [ELSE expr3]

DO may be used in place of THEN. expr1, which must be of type scalar, is evaluated and if its result value is true (odd), then expr2 is evaluated and its result becomes the result of the IF operator. Otherwise, expr3, if present, is evaluated and becomes the result value. If expr3 (which may include another IF operator) is not present, then the ELSE value is unspecified. The result type of the THEN and ELSE operands is specified in Appendix C. Only if an IF operation result is used as an operand of a larger expression can a type check occur. Examples:

```
SCALAR   S, T, X
VECTOR   V, D
```

T ← (IF X THEN S ELSE D)	illegal, THEN and ELSE types don't match
IF X THEN V ← D ELSE T ← S	OK
T ← (IF X THEN S)	illegal, ELSE value undefined
IF X THEN T ← S	OK
T ← (IF THEN S ELSE GOTO L)	OK, GOTO result cannot be used

← is the assignment operator. It has the same precedence as the indirection operator ('$') for its left operand and ranks just below IF for its right operand. The right operand is evaluated, and its result value becomes the result value of the assignment operation, and also becomes the value of the element in memory specified by the result reference of the left operand. The left operand must be a memory reference expression which references variable memory. If the element size specified by the left operand is a partial word of n bits then only the right n bits of the 32 bit right operand result value are stored in memory. The table in Appendix C details the result type and number of words of value stored for the assignment operation as a function of the types of the left and right operands. Note that if both operands are of type any then only 1 word is stored, that is the operands are assumed to be scalar.

OR is the Boolean or . It is a scalar operator. The result value is 1 if either operand is true; otherwise, the result value is Ø . The right operand is not evaluated if the left operand is true.

Example: 4 < 5 OR F(X) result: 1
F(X) not evaluated

AND is the Boolean and . It is a scalar operator. The result value is 1 if both the operands are true; otherwise, the result value is Ø . The right operand is not evaluated if the left operand is false.

Example: 8 < 6 OR F(X) result: Ø
F(X) not evaluated

NOT — is the Boolean not. It is a scalar operator. Its result value is 1 if its evaluated operand is false, otherwise its result value is ∅ .

= # < <= > >= — are the relations equal to, not equal to, less than, not greater than, greater than, not less than, respectively. They are scalar operators. Each relational operator evaluates its operands and then performs the test. If the relation is true, then the result value is 1 ; otherwise, the result value is ∅ .

REM — computes the mathematical remainder after division. It is a scalar operator. The magnitude of A REM B is |A| modulo |B| . The sign of the result is the sign of B .

+ - — perform 32-bit two's complement integer addition and subtraction, respectively. They are scalar operators.

* / SHIFT CYCLE — '*' and '/' perform 32-bit two's complement integer multiplication and division, respectively. SHIFT shifts the 32 bits of the left operand the number of places indicated by the two's complement right operand. A positive right operand specifies a right shift and a negative left shift. Vacated bits are replaced by zeros. CYCLE is the same but performs 32-bit cyclic shifts; that is, the bits that spill off one end are shifted in at the other end. All these operators are scalar operaotrs.

+ -
GOTO
RETURN
SRETURN
FRETURN
EXIT

are the unary operators. Unary '+' is ignored. Unary '-' is a scalar operator and computes the two's complement of its operand. GOTO interprets the value of its operand as an instruction address within the current function body. Control is transferred to this address. In general, to be meaningful, the value of the operand should be the value of a label. The operand must be of type scalar. Only memory reference expressions are legal operands to a GOTO operator.

Examples:

```
GOTO L              (where L is a label)
GOTO DISP[I]        (where DISP is a vector whose
                     elements have been initialized to the
                     values of labels)
```

RETURN , SRETURN , and FRETURN cause control to return to the calling function from the current function. A function may return a single value and a predicacy condition (success or failure). Normally, the single operand, if present, is evaluated and becomes the return value. If not present, then the return value is undefined. RETURN and SRETURN specify return with a success-predicacy condition. The table in Appendix C indicates which combinations of operand type and return type specified in the function head are illegal for SRETURN and RETURN . FRETURN specifies return with a failure-predicacy condition. The operand of FRETURN must be of type scalar.

EXIT specifies control to leave the current control block. If the single argument, which must be a label name, is present, then control is transferred to the statement following the last statement of the control block labelled by the operand. A control block is named by a label on the IF , FOR , or WHILE statement which begins the control block. It is a program error if the EXIT operator is not textually enclosed within the specified control block. If the operand of the EXIT operator is not present, then the innermost control block is exited. It is a program error if there is no enclosing control block.

$ @ Unary '$' is the indirection operator. Its single operand must be of type reference. The result value is the value in memory described by the reference. If specified in the reference, partial word values are sign extended. The result size is the same size specified by the reference. A runtime error occurs if indirection is applied to a reference to a multi-word value. If the operand is a simple name declared as a reference, then the result type is the type specified in the name declaration, otherwise the result is of type any. The result reference is just the operand result value. Unary '@' is the reference operator. Its operand must be a memory reference expression. The result type is reference and its two-word result value is a reference to the value in memory indicated by the operand.

Warning! References to local values will not remain valid after a return to the calling function.

\$. @ Binary '$' , '.' and '@' are the field access or tailing operators. For these operators, the right operand must be of type scalar and is interpreted as a field selector. For the '.' operator, the left operand, which must be of type reference, specifies the location in memory of a value. For the '$' operator, the left operand is a value which may be of any size or type. For both '.' and '$' the result value is the value of the field selected by the field selector of the value specified by the left operand. If specified in the field selector, partial word fields are sign extended. A runtime error occurs if the selected field is not within the value; e.g., the displacement is greater than the size of the value. The result size is the size of the selected field. If the right operand is a simple name declared in a FIELD statement then the result type is the type specified in the name declaration, otherwise the result is of type any. The result reference is to the field in memory specified by the field reference operation.

For the '@' operator, the left operand must be a value of type scalar. The result value is constructed by placing the value of the left operand into the selected field, i.e., the result value of V @ F is the value of T after (T ← Ø ; T\$F ← V). The result is of type scalar. The selected field must have size not greater than one word.

[] specify the vector subscripting operation. Only one dimensional subscripting is allowed. The operand preceding the square brackets must be of type vector and describes a vector in memory. The operand enclosed in the square brackets is the subscript (or index) and specifies the element within the vector to be referenced. Vector elements are numbered starting with zero or one depending on the lower bound field in the vector descriptors. If the subscript is out of bounds, then a failure action is performed according to the rules given with the function call semantics. The result value is the value of the specified vector element. If specified in the descriptor, the value from a partial-word element sign is extended. The result size is the element size specified in the vector descriptor. If the vector operand is a simple name declared as a vector, then the result type is the element type specified in the name declaration, otherwise the result is of type any. The result reference is to the selected element of the vector.

() when immediately preceded by an operand, specify a function call. The preceding operand specifies the function to be called. Typically, this operand will be a function name (the value of a function name is the function's identification number); however, it may be any memory reference expression of type scalar. In general, to be meaningful, the value of the operand should be obtained from the value of a function name.

Arguments.

Inside the parentheses may be specified a list of actual argument expressions separated by commas. The number and types of the actual arguments must match the number and types of the formal arguments of the function to be called. If the function to be called is specified by its function name, then actual arguments may be ommitted from the actual argument list if default values for the corresponding formal arguments are specified in the FUNCTION statement of the function to be called. The effect is exactly the same as if the default value were inserted directly into the actual argument list.

Example: If F has six formal arguments, then in the call

F(, X,, Y)

the first, third, fifth, and sixth actual arguments are omitted and default values for the corresponding formal arguments must be specified.

All arguments are passed by value; thus, the value of each actual argument expression is assigned as the value of the corresponding formal argument. For each pair of corresponding actual and formal arguments the table in Appendix C indicates the legality of the argument and the number of words of value copied as the argument is passed.

Failure Action.

If a function call returns a failure predicacy (or if a subscripting or ring access operation fails) a failure action is performed. The fail action performed is determined from the <failure> or <fail-action> clause specified with the failing operator according to the following rules:

1) If no <failure> or <failaction> is present (i.e., no colon), then a runtime trap occurs.

   Example: F(X)

2) If only the colon is present then execution will continue as if the operation had succeeded. However, the result will be unspecified.

   Example: IF X=Ø THEN GETF(S:)

3) If a square-bracketed name (which must be a variable) is present following the colon (may be present only for function calls), then the failure return value of the function call is stored in the specified variable.

   Example: F(X:[T])

4) If an <expression> is present after the colon, then the expression is evaluated and its result becomes the result of the operation. If a square-bracketed name is also present, then the failure value store is performed before the expression is evaluated.

   Example: F(X: [T]G(T))

   A[I: FRETURN]

FRONT
REAR
GETF
GETR
PUTF
PUTR

are the ring access operators. They use function syntax. The first operand must be of type vector, and specifies a vector descriptor for the ring to be accessed. The second operand must be of type scalar and specifies the ring selector for the ring access operation. If the second operand is omitted, then the first operand must be a simple name and that name with 'PTR' appended is taken to be the name of the ring selector. If present, the second operand may be any expression for FRONT or REAR but must be a simple variable name for GETF, GETR, PUTF, or PUTR . If the ring access operation fails, then a failaction is performed according to the rules given with the function call semantics. All six ring operators will fail if either pointer in the ring selector is out of bounds of the vector to be accessed. FRONT, REAR, GETF, and GETR will fail if the ring selector indicates an empty ring. PUTF and PUTR will fail if the ring selector indicates a full ring. PUTF and PUTR may only be used as the principal operator of the left operand of an assignment operator. GETF and GETR may be used in any context except as the principle operator of the left operand of an assignment operator. The result value for FRONT and GETF is the first element in the ring and for REAR and GETR it is the last element in the ring. GETF and GETR have the side effect of updating the ring selector to remove the referenced element from the ring. PUTF and PUTR have no result value and yield as result reference the element just before the front ring element or just after the rear ring element, respectively, updating the ring selector to include the referenced element. For all six ring access operators, the

result size is the element size from the vector descriptor. If the first operand is a simple name declared as a vector, then the result type is the type specified in the name declaration, otherwise, the result is of type any. The result reference for FRONT, REAR, GETF, and GETR is to the element referenced at the front or rear of the ring.

AND
OR
EOR — when used in function syntax specify the bitwise logical operators. They are scalar operators. The result value for AND is the end of the corresponding bits of the operands, for OR it is the bitwise inclusive or, and for EOR it is the bitwise exclusive or.

ABS — give as a result value the absolute value of its operand interpreted as a 32 bit two's complement number. It is a scalar operator.

CODE — can take one or more operands. The operands must all be constant expressions of type scalar. The value of the operands is compiled as consecutive instructions in the object code of the program. Thus operator makes it possible to explicitly specify values to be stored in the code segment and makes it possible to compile code that cannot be generated in any other way in SIMPLE.

9. Statements.

Syntax:

statement = scalarstat / vectorstat / constantstat / ringstat /
macrostat / referencestat / descriptorstat /
equivalencestat / stringstat / fieldstat / functionstat /
endstat / execstat

Statements are of five types:

(1) declaration statements

SCALAR, CONSTANT, VECTOR, RING, MACRO, DESCRIPTOR, STRING, REFERENCE, FIELD, EQUIVALENCE

(2) function block statements

FUNCTION, END

(3) control block statements

IF, ELSEIF, ELSE, ENDIF, FOR, WHILE, ENDFOR, ENDWHILE

(4) expression statements

(5) miscellaneous statements

BREAK

The control block, expression, and miscellaneous statements are collectively referred to as executable statements.

9.1 Block Organization

A function block must begin with a FUNCTION statement, any block may optionally end with an END statement. The body of the function follows the FUNCTION statement. All declaration statements must textually precede all executable statements in the function body.

The global declaration block must consist entirely of declaration statements.

9.2 Executable Statements and Labels

Syntax:

```
execstat = arb(name':') [ifstat / elseifstat / elsestat / endifstat /
           forstat / whilestat / endforstat / endwhilestat / breakstat /
           expressionstat]
```

An arbitrary number of label names may be specified at the beginning of any executable statement. A label is not legal on a declaration statement or a function block statement. A name specified as a label is declared as a scalar constant local to the current function block. The value of the constant is the function base relative instruction address of the first instruction in the current statement.

Unlike all other names in SIMPLE, except function names, label names may be used in statements before they are declared. It is a program error if an undeclared name is used and not subsequently declared as a label within the function block. Furthermore, it is not legal to declare a name as a label if it is globally declared. A label name may be used in the following cases:

(1) In any executable statement (except an operand of CODE operator) in any context where a scalar constant is legal.

(2) In initialization lists for VECTOR, RING and STRING declarations and as the operand of an operator. In these cases, the label name must be used either as a simple expression or as the left operand of an '@' field tailing operator. In this latter case, (which is a kludge feature to facilitate table construction) the field selector must specify a 16 bit field that lines up on

the half-word boundary and the bits specified by the field must not be modified by any other operator in the expression.

(3) As the operand of an EXIT operator.

Examples:

```
L1:      A ← B

L2:L3:   GOTO L4
```

10.1 SCALAR Statements

Syntax:

scalarstat = SCALAR sep(vardec, ',')

vardec = name [TO type]

A SCALAR statement specifies a list of names to be declared as variables of type scalar. The <type>, if present, specifies the type of fields accessed if the name is used as a field selector in a field tailing operation.

Example:

```
SCALAR A, B, C, TO VECTOR
```

10.2 CONSTANT Statement

Syntax:

constantstat = CONSTANT sep(condec, ',')

condec = name '←' conditional

A CONSTANT statement specifies a list of names to be declared as scalar constants. The right hand side of each left arrow must be a constant expression which specifies the value for the declared constant name.

Example:

```
CONSTANT  X←3, Y←X+7
```

10.3 VECTOR Statement

Syntax:

```
vectorstat = VECTOR vecdec

vecdec     = name [vecspec] ['←' initlist / ',' vecdec]

vecspec    = '[' [vecdesc] [type] ']'

vecdesc    = conditional1 [SIGNED] [ONE\BASED] [EXTERNAL conditonal]
             [conditional2 (WORD/BIT)]

type       = SCALAR / VECTOR / REFERENCE / DESCRIPTOR / RING / STRING / ANY
             SCALARS / VECTORS / REFERENCES / DESCRIPTORS / RINGS / STRINGS / ANYS

initlist   = sep(conditional, ',')
```

A VECTOR statement specifies a list of names to be declared as type vector. Furthermore, for each name the following information may be specified:

(1) a vector descriptor for the vector,

(2) space to be allocated for the vector,

(3) initial values for the vector elements,

(4) a type for the vector elements.

Syntactically, each name declared may be optionally followed by a specification enclosed in square brackets (<vecspec>) and the last name in the list may be optionally followed by an initialization list (<initlist>). Inside the square brackets may be specified vector descriptor information (<vecdesc>) and a vector element type (<type>).

Element type specification.

If present, (<type>) specifies the type of the values stored in the vector. If not present then SCALARS is assumed (note exception below if initialization list is present).

Variable vector specification.

If neither the vector descriptor information nor the initialization list is present, then the name is declared as a variable and no space is allocated for the vector. In this case, vector space allocation and descriptor set-up must be done by the program at runtime.

Vector descriptor specification.

If the vector descriptor information (<vecdesc>) is present, then the name is declared as a constant of type vector and space is allocated for the vector. The <conditional> expressions must be constant, <conditional1> specifying the vector size in elements and <conditional2> the element size. Legal element sizes are 8-BIT, 16-BIT and x-WORD where x is in the range 1 to 127. If no element size is specified, then one word elements are assumed if the vector elements are of type scalar and 2-word elements are assumed if the vector elements are of type reference, or vector. Furthermore, if the element type is vector or reference, then the element size, if present, must be 2 words. If SIGNED is present, then partial word elements when fetched will be sign extended to 32-bit values. Otherwise, the high order bits are zero. If ONE\BASED is present then indices in the vector begin at one instead of zero.

External vector specification.

If the EXTERNAL clause is present in the vector descriptor information then the name is declared as a vector constant for an external segment vector. The expression after EXTERNAL, which must be constant in the range Ø-177777B, specifies the capability number. The base address is taken to be zero. External vectors may not be initialized.

Vector of initialization.

If the initialization list is present then vector storage for the vector is allocated as constant. Default values for vector size (number of elements), vector element size, and vector element type are provided if these specifications are not present according to which form of initialization list is present. If a vector element size and/or a vector element type is present then it must match the corresponding default size or type for the specified form of initialization list. Initialization lists may be in one of four forms:

(1) A list of constant expressions of type scalar. The successive values in the list specify the values of successive elements of the constant vector. If no vector size is specified, then the default size matches the length of the initialization list. If a vector size is specified, then it must be at least as big as the list length. The remaining values will be initialized to zero. The element size, if present, must not be larger than one word. The element type, if present, must be scalar. The default vector element size and vector element type are one word and scalar, respectively.

(2) A list of expressions of type vector. The successive vector elements are initialized to the vector descriptors specified in the initialization list. The list elements must be previously declared names of type vector constant. The vector size, if present, must exactly match the length of the initialization list. The element size, if present, must be 2 words. The element type, if present, must be

vector. The default values for vector element size and vector element type are 2 word and vector, respectively.

(3) A list of constant expressions of type reference. The successive vector elements are initialized to the references specified in the initialization list. The vector size, if present, must exactly match the length of the initialization list. The list elements must be string constants or names previously declared in a STRING statement. The element size, if present, must be 2 words and the element type if present, must be reference. The default values for vector element size and vector element type are 2 word and vector respectively.

(4) A single string constant. The successive vector elements are initialized to the character code values for the successive characters of the string constant. The vector size, if present, must exactly match the size of the string constant. The element size, if present, must be 8 bits. The element type, if present, must be scalar. The default values for vector element size and vector element type are 8 bit and scalar, respectively.

Example:

```
VECTOR A,B[7], C[128  2 WORD VECTORS]
VECTOR D[6 16 BIT] ← Ø, 1, 2, 3, 4, 5
VECTOR S ← "ABCDEF"
```

10.4 RING Statement

Syntax:

ring = RING vecdec

The syntax and semantics of a RING statement are identical to a VECTOR statement except that:

(1) an extra element is allocated in the space for the ring, since a "full" ring actually has one unused element (see SIPU manual), and;

(2) for each name in a RING statement, a scalar name is declared with a name which appends 'PTR' to the vector name. This scalar name is intended for use as the selector for ring access operations. The selector name is declared as a constant if an initialization list is specified for the name, the value of the constant being a selector for all the elements in the ring. A variable selector is declared if no initialization list is present.

10.5 STRING Statement

Syntax:

stringstat = STRING vecdec

A STRING statement specifies a list of names to be declared as constants of type reference. The constant reference value is to a three word value which is interpreted as a "string descriptor." The first two words of a string descriptor are a vector descriptor for the string and the third word is a ring selector. The syntax for STRING statements is exactly the same as VECTOR statements. The rules for vector descriptor specification, vector initialization, and vector element type specification are exactly the same as for VECTOR statements except that an extra element is always added to the vector size and that the default element size is 8 bits.

The string descriptor is allocated and initialized according to the following rules:

(1) If no vector descriptor information (<vecdesc>) and no initilization list is present then an uninitialized 3 word string descriptor is allocated in global memory.

(2) If only the (<vecdesc>) is present (no initialization list) then a 3 word string descriptor is allocated in global memory and the first two words are initialized to a vector descriptor for a vector allocated in global memory. The ring selector word is not initialized.

(3) If an initialization list is present, then a 3 word string descriptor is allocated in constant memory. The first two words are initialized to a vector

descriptor for the vector initialized in constant memory and the third word is initialized to a ring selector for all of the constant vector.

10.6 REFERENCE Statement

Syntax:

referencestat = REFERENCE sep(vardec, ',')

The REFERENCE statement specifies a list of names to be declared as variable values of type reference. If present, <type> specifies the type of value the reference locates. If not present, type scalar is assumed.

Example:

```
REFERENCE X, Y TO VECTOR, Z
```

10.7 DESCRIPTOR Statement

Syntax:

descriptorstat = DESCRIPTOR spe(vardec, ',')

The DESCRIPTOR statement specifies a list of names to be declared as variables of type descriptor, i.e., type reference or type vector simultaneously. If present, <type> specifies the type of values accessed through the vector descriptor or reference. If not present, then type scalar is assumed.

10.8 FIELD Statement

Syntax:

```
fieldstat = FIELD  sep(fldspec, ',')
fldspec   = name  '(' [SIGNED] conditional1 [':'  conditional2 ','
              conditional3 / THRU  conditional] [ TO type] ')'
```

A FIELD statement specifies a list of names to be declared as scalar constants and initialized as field selectors. The three <conditional> expressions must be constant. The <conditional1> specifies a displacement. The <conditional2> and <conditional3>, if present, specify the left and right bits of the partial word field inclusive. The expression after THRU , if present specifies the displacement of the last word of a multi-word field. If only one expression is present, a full word field is assumed. The high order bit of a scalar is bit Ø and the low order bit is bit 31. If present the <type> specifies the type of value accessed by the field selector. If the specified type is for vectors or references then the specified selector must be for a two word field. If the <type> is not present then type scalar is assumed.

Example:

```
FIELD F(3), G(Ø:17, 31), H(2 THRU 4)
```

10.9 EQUIVALENCE Statement

Syntax:

```
equivalencestat = EQUIVALENCE  sep(eqvdec, ',')
eqvdec          = name '=' name '[' conditional ']'
```

An EQUIVALENCE statement specifies a list of names which are to be declared as equivalent to the values specified on the right hand side of the equal signs. The right hand side specifies an element of a vector. The right hand name must be previously declared as a vector with space allocated in global memory and with element size of one or two words. The name will be declared with the same type as the values in the vector elements. The <conditional> inside the square brackets must be a constant expression which specifies the index of the vector element.

10.10 MACRO Statement

Syntax:

macrostat = MACRO name ['(' [dummylist] ')'] '←' macrobody

dummylist = sep(name, ',')

The MACRO statement specifies the definition of the macro being declared. The <macrobody> may be any arbitrary string of primaries up to but not including the statement terminating semi-colon or carriage return. An exclamation point (!) within a <macrobody> is identical to a semi-colon except that it does not terminate the macro definition. This allows macro definitions that expand into several statements. Macro calls within a macro definition body are not expanded at definition time but at call time. The <dummylist> , if present, specifies a list of names which, when used within the macro body, serve to mark the places where the actual arguments of a macro call should be substituted. The use of a name as a dummy argument in no way interferes with any other use of the same name within a SIMPLE program.

Examples:

```
MACRO DOUBLE(XX) ← (XX)*2
MACRO ERROR (XX) ← (EFUNC(XX) & FRETURN)
```

Macro Calls

Syntax:

macrocall = name [actuallist]

actuallist = '('[sep(actualargument, ',')]')'

A macro call occurs when a name previously declared as a macro occurs anywhere within a statement. If the macro is defined with a <dummylist>, then the call must include an <actuallist> and the number of actual arguments

must exactly match the number of dummy arguments. An actual argument is any string of primaries balanced with respect to parentheses, not containing a semi-colon or carriage return primary, and delimited by a comma or a colon not enclosed in inner parentheses. The effect of the macro call is that the macro body, with actual arguments substituted for dummy arguments, replaces the macro call in the statement. Macro calls are expanded in a strictly left to right scan of the statement. After a macro call, the scan continues with the first primary of the substituted definition.

Note that substitution is done by primaries. Arbitrary string substitution is not possible.

11. Function Block Statements

11.1 FUNCTION Statement

Syntax:

functionstat = FUNCTION name '(' [formallist] ')' [RETURNING type]

formallist = sep(formal, ',')

formal = [type] name [TO type] [':' conditional]

A FUNCTION statement must be the first statement of a function block. It specifies the function name, the type of value the function returns, if successful, and a list of formal arguments. The function name specified is declared as a global, scalar constant with a value called the function number which is different from all other function numbers. If the RETURNING clause is present, then the <type> specifies the type of value returned by the function on success return (default type is scalar). Fail return values are always of type scalar. Each name in the formal argument list is declared as a local variable with type specified by the prededing <type> (default type is scalar). The TO clause, if present, specifies the type of value referenced in memory if the formal argument is used as a vector descriptor, reference, or field selector (default type is scalar). If present, the <conditional> after the colon, which must be a constant expression, specifies a default value to be assigned to the formal argument (which must be of type scalar) if no actual argument is provided in a function call.

Example:

```
FUNCTION  F(X, Y:1,  VECTOR Z TO RINGS) RETURNING REFERENCES
```

11.2 END Statement

Syntax:

endstat = END

The END statement may be optionally used as the last statement of a global declaration or function block

12. Control Blocks

Control blocks allow specification of a sequence of statements that as a group may be executed out of linear sequence. There are two kinds of control blocks:

(1) conditional control blocks which specify a sequence of statements which is executed only if a specified expression evaluates to a true (or false) value;

(2) iteration control blocks which specify a sequence of statements which is executed repeatedly while a specified condition is true.

12.1 Conditional Control Blocks

Syntax:

ifstat = IF or (DO / THEN)

elseifstat = ELSEIF or (DO / THEN)

elsestat = ELSE [DO]

endifstat = ENDIF

A conditional control block takes the form

```
IF statement
        .
        . ifbody
        .
ELSEIF statement (Ø or more ELSEIF's allowed)
        .
        . elseifbody
        .
ELSE statement (ELSE may be omitted)
        .
        . elsebody
        .
ENDIF statement
```

The dots may be any sequence of executable statements balanced and well nested with respect to IF and ENDIF, FOR and ENDFOR and WHILE and ENDWHILE statements. The statements of an ifbody or elseifbody are executed only if it is the first body in the structure for which the preceding <or> is true. The elsebody is executed if none of the <or>'s (which must have results of type scalar) are true. After the body is executed control transfers to the statement after the ENDIF . In SIMPLE a value is defined as true if it is odd and false if it is even. Systematic indentation of nested control blocks is strongly recommended.

12.2 Iteration Control Blocks

Syntax:

```
forstat        = FOR forclause (DO / THEN)

forclause      = [reporder] name '←' (whclause / toclause)

whclause       = conditional1 [',' conditional2] WHILE conditional3

toclause       = conditional1 [BY conditional2] TO conditional3

whilestat      = WHILE whileclause (DO / THEN)

whileclause    = [reporder] conditional

reporder       = TBU / TUB / BTU / BUT / UTB / UBT

endforstat     = ENDFOR

endwhilestat   = ENDWHILE
```

An iteration control block takes the form

```
FOR or WHILE statement
        .
        . forbody
        .
ENDFOR or ENDWHILE
```

The dots may be any sequence of executable statements well nested with respect to other control blocks. In general, every repetition consists of three actions.

- Test : the test expression is evaluated and if false repetition ends and control is transferred to the statement after the ENDFOR or ENDWHILE
- Body : the statements of the forbody are executed
- Update : the update expression is evaluated and assigned to the iteration control variable.

There are six possible orders of execution of the three actions within a single repetition: BTU, BUT, TBU, TUB, UTB, UBT. In addition to these actions an initial value for the iteration control variable can be specified.

If present <reporder> specifies the order actions are performed within a repetition. If not present, then TBU is assumed. However, if a whclause is present with no initialization then UTB is assumed.

In a <forclause> the name specifies the iteration control variable. If the name is already declared it must be a scalar variable, otherwise it is declared locally as a scalar variable.

In a <whclause> , if <conditional2> is present, then <conditional1> specifies the initial value for the iteration control variable and <conditional2> is the update expression. If <conditional2> is not present, <conditional1> is the update expression and no initialization is specified. <conditional3> is the test expression.

In a <toclause>, <conditional1> specifies the initial value of the iteration control variable, <conditional2>, which is reevaluated on each repetition, specifies the value to be added to the iteration control variable as the update action, and <conditonal3> specifies the test expression. Iteration will continue while the value of the control variable is not greater than the value of the test expression, unless the increment is a negative constant,in which case while the control variable is not less than the test expression. If <conditional2> is not present, an increment of one is assumed.

In the <whileclause> , the <conditional> specifies the test expression. No initialization and update action is specified, thus the six repetition orders actually specify only two different cases, body-test and test-body.

13. Expression Statements

Syntax:

expressionstat = expression

An expression statement consists solely of an expression. The expression is evaluated and its result value is left in the accumulator of the SIPU.

14. Miscellaneous Statements

14.1 BREAK Statement

Syntax:

breakstat = BREAK

The BREAK statement causes a breakpoint to be set in the program.

Appendix A

Character Codes

Code	Character	Code	Character	Code	Character	Code	Character
ø	blank	20	ø	40	@	60	P
1	!	21	1	41	A	61	Q
2	"	22	2	42	B	62	R
3	#	23	3	43	C	63	S
4	$	24	4	44	D	64	T
5	%	25	5	45	E	65	U
6	&	26	6	46	F	66	V
7	'	27	7	47	G	67	W
10	(	30	8	50	H	70	X
11	)	31	9	51	I	71	Y
12	*	32	:	52	J	72	Z
13	+	33	;	53	K	73	[
14	,	34	<	54	L	74	\
15	-	35	=	55	M	75	]
16	.	36	>	56	N	76	↑
17	/	37	?	57	O	77	←
						155	Carriage return

Codes in octal

Appendix B

Rules for Types for Expression Operators

operator	left operand type	right operand type	result type
&	all types OK	all types OK	same as right operand
WHERE	all types OK	all types OK	same as left operand
IF		scalar	given by Appendix C if ELSE present, otherwise is same type as THEN operand
THEN		all types OK	
ELSE		must match THEN type according to Appendix C	
←	---------- See Appendix C ----------		
OR AND (Boolean) = # < <= > >= REM + - * / CYCLE SHIFT	scalar	scalar	scalar
RETURN SRETURN FRETURN		must match return type in FUNCTION statement according to Appendix C	undefined
EXIT		label name	undefined
+ - NOT (unary)		scalar	scalar
GOTO		scalar	undefined
$ indirection		reference	specified by name declaration or any
@ reference		all types OK	reference

table continues

operator	left operand type	right operand type	result type
. field	reference	scalar	specified by FIELD statement or any
$ field	all types OK	scalar	specified by FIELD statement or any
@ field	scalar	scalar	scalar
[]	vector	scalar	specified by name declaration or any
() function call	scalar	arguments must be of type specified in FUNCTION statement	specified by FUNCTION statement
FRONT REAR GETF GETR PUTF PUTR	vector	scalar	specified by name declaration or any
AND OR EOR (logical)	scalar	scalar	scalar
ABS		scalar	scalar

Appendix C

Type Semantics

first operand type

second operand type	Scalar	Vector	Reference	Descriptor	Any
Scalar	scalar 1 word	illegal	illegal	illegal	scalar 1 word
Vector	illegal	vector 2 words	illegal	vector 2 words	vector 2 words
Reference	illegal	illegal	reference 2 words	reference 2 words	reference 2 words
Descriptor	illegal	vector 2 words	reference 2 words	descriptor 2 words	descriptor 2 words
Any	scalar 1 word	vector 2 words	reference 2 words	descriptor 2 words	any 1 word

This table specifies the type semantics for the following four contexts within expressions. Some combination of operand types are illegal. For the remaining combinations, the result type and the number of words of value to copy are specified.

1) Assignment operator.

first operand = left operand of '←'

second operand = right operand of '←'

If "illegal" then a compile time type check error will occur.

2) Function Argument Passing.

first operand = formal argument

second operand = actual argument

If "illegal" then a compile time type check error will occur.

3) Returning value successfully from function.

first operand = operand of SRETURN or RETURN

second operand = type specifies in FUNCTION statement

If "illegal" then a compile time type check error will occur.

4) IF operator when ELSE clause present.

first operand = THEN operand

second operand = ELSE operand

If "illegal" a compile time error will occur only if the result of the IF operator is used in a larger expression.

Appendix D

Reserved Words

ABS
AND
ANY
ANYS
BIT
BREAK
BTU
BUT
BY
CODE
CONSTANT
CYCLE
DESCRIPTOR
DESCRIPTORS
DO
ELSE
ELSEIF
END
ENDFOR
ENDIF
ENDWHILE
EOR
EQUIVALENCE
EXTERNAL
EXIT
FIELD
FOR
FRETURN
FRONT
FUNCTION
GETF
GETR
GLOBAL
GOTO
IF
MACRO
NOT
ONE\BASED
OR
PUTF
PUTR
REAR
REFERENCE
REFERENCES
REM
RETURN
RETURNING
RING
RINGS
SCALAR
SCALARS
SHIFT
SIGNED
SRETURN
STRING
STRINGS
TBU
THEN
THRU
TUB
UBT
UTB
VECTOR
VECTORS
WHERE
WHILE
WORD

Appendix E

Constant Expressions

When most operators are used as principle operators, an expression can be evaluated as a constant expression if all its operands are themselves constant expressions. All the exceptions to this rule are listed below.

1) Operators that yield unspecified results may not be evaluated as constant expressions. This includes the following operators: GOTO, RETURN, SRETURN, FRETURN, and EXIT.

2) The assignment and function call operators may not be constant evaluated.

3) Operators that reference memory may not be evaluated as constant expressions. This includes the following operators: indirection, field access operators, ring access operators, and subscripting.

4) The IF operator is evaluated as a constant expression if (a) the IF operand is constant and true valued and the THEN operand is constant or (b) the IF operand is constant and false valued and the ELSE operand is present and constant.

5) The reference operator ('@') is evaluated as a constant if its operand is (a) a constant expression or (b) a simple global variable name.