Space Flight Operations Contract

HAL/S LANGUAGE SPECIFICATION

PASS 32.0/BFS 17.0

November 2005

DRD - 1.4.3.8-a

Contract NAS9-20000

HAL/S LANGUAGE SPECIFICATION

Approved by

Original Approval Obtained
Barbara Whitfield, Manager
HAL/S Compiler and Application Tools

Original Approval Obtained
Monica Leone, Director
Application Tools Build and Data Reconfiguration

DRD - 1.4.3.8-a
Contract NAS9-20000

Revision Log

The HAL/S Language Specification has been revised and issued on the following dates(s):[1]

Issue	Revision	Date	Change Authority	Sections Changed
29.0/14.0		03/05/99	CR13043	- pp. Ix, x, xi, xii, iii - p. 1-2 - pp. 2-1, 2-3, 2-4, 2-6, 2-7, 2-8 - pp. 3-1, 3-2, 3-3, 3-6, 3-8, 3-9, 3-10, 3-11, 3-12, 3-13 - pp. 4-1, 4-3, 4-4, 4-5, 4-7, 4-8, 4-9, 4-10, 4-11, 4-13, 4-14, 4-15, 4-16, 4-17, 4-18, 4-19, 4-20 - pp. 5-1, 5-2, 5-3, 5-4, 5-6, 5-7, 5-8, 5-9, 5-10, 5-11, 5-12, 5-14 - pp. 6-1, 6-2, 6-4, 6-6, 6-8, 6-12, 6-13, 6-14, 6-15, 6-16, 6-18, 6-19, 6-20, 6-21, 6-22, 6-24, 6-25, 6-26, 6-27, 6-28 - pp. 7-1, 7-2, 7-3, 7-4, 7-5, 7-6, 7-7, 7-8, 7-9, 7-11, 7-12, 7-13, 7-14, 7-15 - pp. 8-3, 8-5, 8-8, 8-9, 8-10 - pp. 9-1, 9-2, 9-4 - pp. 10-3, 10-4, 10-7, 10-8, 10-9, 10-11, 10-13, 10-14, 10-15, 10-16, 10-18 - pp. 11-2, 11-4, 11-6, 11-10, 11-12, 11-13, 11-14, 11-16, 11-17, 11-18, 11-19, 11-20, 11-21, 11-23, 11-24, 11-25, 11-26, 11-27, 11-28, 11-29, 11-30 - pp. A-3, A-4, A-5, A-6, A-7, A-8, A-9 - pp. C-1, C-2, C-4, C-5 - p. D-1 - p. E-1 - p. F-1 - p. H-2, H-3

1. A table containing the Revision History of this document prior to the USA contract can be found in Appendix J.

Revision Log Cont'd

Issue	Revision	Date	Change Authority	Sections Changed	
			CR12935A	4.5	- p. 4-13
			CR13813	7.4	-p. 7-6
			CR13956	4.8	-p. 4-19
			CR14062	3.7.2 3.7.3 Appx D Index	-p. 3-9 -p. 3-9 -p. D-1 -pp. INDEX-1 to INDEX-10
			CR14215B	6.5.2 Appx D	-p. 6-23 -p. D-1
			CR14216A		Preface
			DR109063	App. C	- p. C-4
			DR109083	6.1.1	- p. 6-3
30.0/15.0		03/27/00	CR12711	11.4.1	- p. 11-12
			CR13212	4.5 11.4.3	- p. 4-12 - p. 11-15
			CR13217	11.4.2 App. A App. G Index	- p. 11-15 - pp. A-1 to A-6 - pp. G-1 to G-9 - pp. INDEX-1 to INDEX-4
			CR13236	4.5 7.4	- p. 4-13 - p. 7-6
31.0/16.0		09/07/01	CR13454	 7.4 Appx H	cover pages, preface -p. 7-6 -p. H-2, H-3
			CR13350	3.7.4	-p. 3-11
			CR13220	6.3	-p. 6-17
			CR13336	Appx F	-p. F-2
			DR111379	Appx C	-p. C-5

Revision Log Cont'd

Issue	Revision	Date	Change Authority	Sections Changed
32.0/17.0		11/05	CR13538	7.4 -p. 7-6 11.4.1 -p. 11-11 11.4.10 -p. 11-23
			CR13652	7.4 -p. 7-6 11.4.8 -p. 11-19 11.4.9 -p. 11-21
			CR13704A	6.4 -p. 6-18 7.4 -p. 7-6
			CR13754A	6.4 -p. 6-19 7.4 -p. 7-6
			CR13813	7.4 -p. 7-6
			CR13956	4.8 -p. 4-19
			CR14062	3.7.2 -p. 3-9 3.7.3 -p. 3-9 Appx D -p. D-1 Index -pp. INDEX-1 to INDEX-10
			CR14215B	6.5.2 -p. 6-23 Appx D -p. D-1
			CR14216A	Preface

List of Effective Pages

The current status of all pages in this document is as shown below:

Page No.	Change No.
All	32.0/17.0

Preface

The *HAL/S Language Specification* was developed by Intermetrics, Inc., and is currently maintained by the HAL/S project of United Space Alliance.

The HAL/S programming language accomplishes three significant objectives:

- increased readability, through the use of a natural two-dimensional mathematical format;
- increased reliability, by providing for selective recognition of common data and subroutines, and by incorporating specific data-protect features;
- real-time control facility, by including a comprehensive set of real-time control commands and signal conditions.

Although HAL/S is designed primarily for programming on-board computers, it is general enough to meet nearly all the needs in the production, verification, and support of aerospace and other real-time applications.

The design of HAL/S exhibits a number of influences, the greatest being the syntax of PL/1 and ALGOL, and the two-dimensional format of MAC/360, a language developed at the Charles Stark Draper Laboratory. With respect to the latter, fundamental contributions to the concept and implementation of MAC were made by Dr. J. Halcombe Laning of the Draper Laboratory.

The primary responsibility is with USA, Department, 01635A7.

Questions concerning the tecnical content of this document should be directed to Danny Strauss (281-282-2647), MC USH-635L.

‘This page intentionally left blank.’

Table of Contents

LIST OF FIGURES

List of Tables

This page is intentionally left blank.

1.0 INTRODUCTION

HAL/S is a programming language developed by Intermetrics, Inc., for the flight software of NASA programs. HAL/S is intended to satisfy virtually all of the flight software requirements of NASA programs. To achieve this, HAL/S incorporates a wide range of features, including applications-oriented data types and organizations, real-time control mechanisms, and constructs for systems programming tasks.

As the name indicates, HAL/S is a dialect of the original HAL language previously developed by Intermetrics. Changes have been incorporated to simplify syntax, curb excessive generality, or facilitate flight code emission.

1.1 Purpose of the Document.

This document constitutes the formal HAL/S Language Specification, its scope being limited to the essentials of HAL/S syntax and semantics. Its purpose is to define completely and unambiguously all aspects of the language. The Specification is intended to serve as the final arbiter in all questions concerning the HAL/S language. It will be the purpose of other documents to give a more informal, tutorial presentation of the language, and to describe the operational aspects of the HAL/S programming system.

1.2 Review of the Language.

HAL/S is a higher order language designed to allow programmers, analysts, and engineers to communicate with the computer in a form approximating natural mathematical expression. Parts of the English language are combined with standard notation to provide a tool that readily encourages programming without demanding computer hardware expertise.

HAL/S compilers accept two formats of the source text: the usual single line format, and also a multiline format corresponding to the natural notation of ordinary algebra.

DATA TYPES AND COMPUTATIONS

HAL/S provides facilities for manipulating a number of different data types. Its integer, scalar, vector, and matrix types, together with the appropriate operators and built-in functions, provide an extremely powerful tool for the implementation of guidance and control algorithms. Bit and character types are also incorporated.

HAL/S permits the formation of multi-dimensional arrays of homogeneous data types, and of tree-like structures which are organizations of non-homogeneous data types.

REAL TIME CONTROL

HAL/S is a real time control language. Defined blocks of code called programs and tasks can be scheduled for execution in a variety of different ways. A wide range of commands for controlling their execution is also provided, including mechanisms for interfacing with external interrupts and other environmental conditions.

ERROR RECOVERY

HAL/S contains an elaborate run time error recovery facility which allows the programmer freedom (within the constraints of safety) to define his own error processing procedures, or to leave control with the operating system.

SYSTEM LANGUAGE

HAL/S contains a number of features especially designed to facilitate its applications to systems programming. Thus it substantially eliminates the necessity of using an assembler language.

PROGRAM RELIABILITY

Program reliability is enhanced when software can, by its design, create effective isolation between various sections of code, while maintaining ease of access to commonly used data. HAL/S is a block oriented language in that blocks of code may be established with locally defined variables that are not visible from outside the block. Separately compiled program blocks can be executed together and communicate through one or more centrally managed and highly visible data pools. In a real time environment, HAL/S couples these precautions with locking mechanisms preventing the uncontrolled usage of sensitive data or areas of code.

1.3 Outline of the Document.

The formal Specification of HAL/S is contained in Sections 3 through 10 of this document. Section 2 introduces the notation to be used in the remainder.

The global structure of HAL/S is presented in Section 3. Data declaration and referencing are presented in Sections 4 and 5, respectively. Section 6 is dedicated to the formation of different kinds of expressions. Sections 7 through 10 show how these expressions are variously used in executable statements.

Section 7 gives the specification of ordinary executable statements such as IF statements, assignments, etc. Section 8 deals with real time programming. Section 9 explains the HAL/S error recovery system and Section 10 the HAL/S I/O capability.

Finally, Section 11 is devoted to system language features of HAL/S.

2.0 SYNTAX DIAGRAMS AND HAL/S PRIMITIVES

In this Specification, the syntax of the HAL/S language is represented in the form of syntax diagrams. These are to be read in conjunction with the associated sets of semantic rules. Sometimes the semantic rules modify or restrict the meaning inherent in the syntax diagrams. Together the two provide a complete, unambiguous description of the language. The syntax diagrams are mutually dependent in that syntactical terms referenced in some diagrams are defined in others. There are, however, a basic set of syntactical terms for which no definition is given. These are the HAL/S "primitives".

This section has two main purposes: to explain how to read syntax diagrams, and to provide definitions of the HAL/S primitives. Various aspects of HAL/S source text which impact upon the meaning of the diagrams are also discussed briefly.

A syntax diagram Cross Reference Table may be found in Appendix A.

2.1 The HAL/S Syntax Diagram.

Syntax diagrams are, essentially, flow diagrams representing the formal grammar of a language. By tracing the paths on a diagram, various examples of the language construct it represents may be created. In this Specification, the Syntax Diagrams, together with the associated semantic rules, provide a complete and unambiguous definition of the HAL/S language. The syntax diagrams are, however, not meant to be viewed as constituting a "working" grammar (that is, an analytical tool for compiler construction).

A typical example of a syntax diagram is illustrated below. Following the diagram, a set of rules for reading it correctly is given. The rules apply generally to all syntax diagrams presented in the ensuing sections.

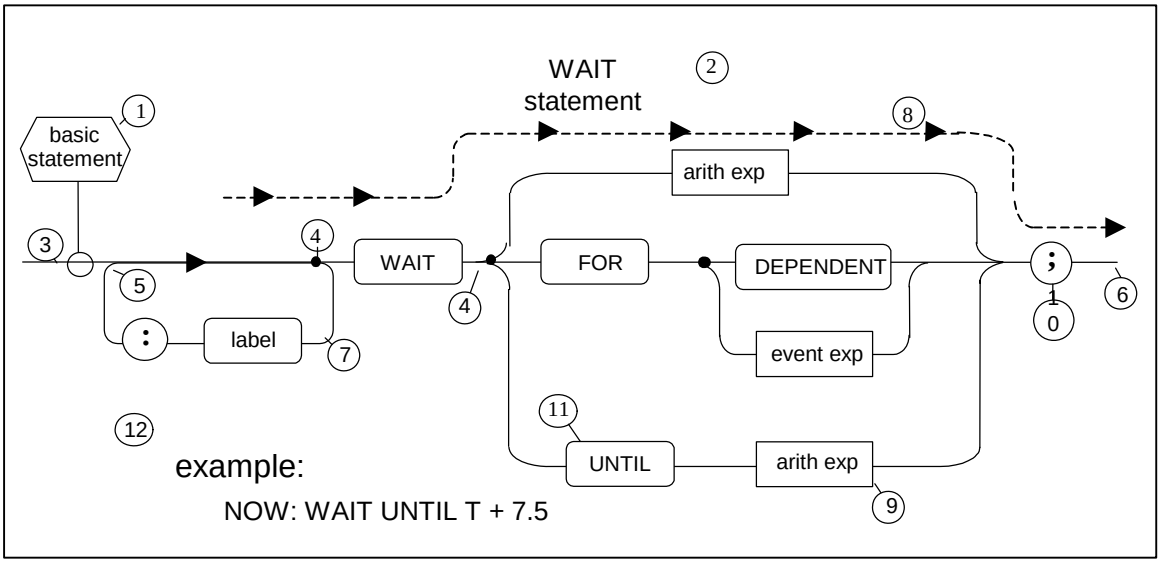


Figure 2-1 WAIT statement (typical syntax diagram)

RULES:

1. Every diagram defines a syntactical term. The name of the term being defined appears in the hexagonal box ①. The title of the syntax diagram is usually a discursive description of the syntactical term. In the case illustrated, the language construct depicted is a particularization of the syntactical term defined (a "WAIT statement" is an example of ①).
2. To generate examples of the construct, the flow path is to be followed from left to right, from box to box, starting at the point of juncture of the definition box ③ , and ending when the end of the path ⑥ is reached.
3. The path is moved along until it arrives at a black dot ④. No "backing up" along points of convergence such as ⑤ is allowed. A black dot denotes that a choice of paths is to be made. The possible number of divergent paths is arbitrary.
4. Potentially infinite loops such as ⑦ may sometimes be encountered. Sometimes there are semantic restrictions upon how many times such loops may be traversed.
5. Every time a box is encountered, the syntactical term it represents is added to the right of the sequence of terms generated by moving along the flow path. For example, moving along the path paralleling dotted line ⑧ generates the sequence "WAIT <arith exp>;" (see Rule 7).
6. Boxes with squared corners, such as ⑨ , represent syntactical terms defined in other diagrams. Boxes with circular ends, such as ⑪, represent HAL/S primitives. Circular boxes, such as ⑩, contain special characters (see Section 2.2).
7. The text accompanying the syntax diagrams, boxes containing lower case names are represented by enclosing the names in the delimiters <>. Thus box ⑨ becomes <arith exp>. Upper case names are reserved words of the language.
8. The example given at ⑫ is an example of HAL/S code which may be generated by applying the syntax diagram (since some boxes, such as ⑨ for example, are defined in other syntax diagrams, reference to them may be necessary to complete the generative process).

2.2 The HAL/S Character Set.

The HAL/S character set consists of the 52 upper and lower case alphabetical characters, the numerals zero through nine, and other symbols. The restricted character set is the set necessary for the generation of constructs depicted by the syntax diagrams. The extended character set includes, in addition, certain other symbols legal in such places as comments and character literals, and is used chiefly for the purpose of compiler listing annotation.

The following table gives a complete list of the characters in the extended set, with a brief indication of their principal usage.

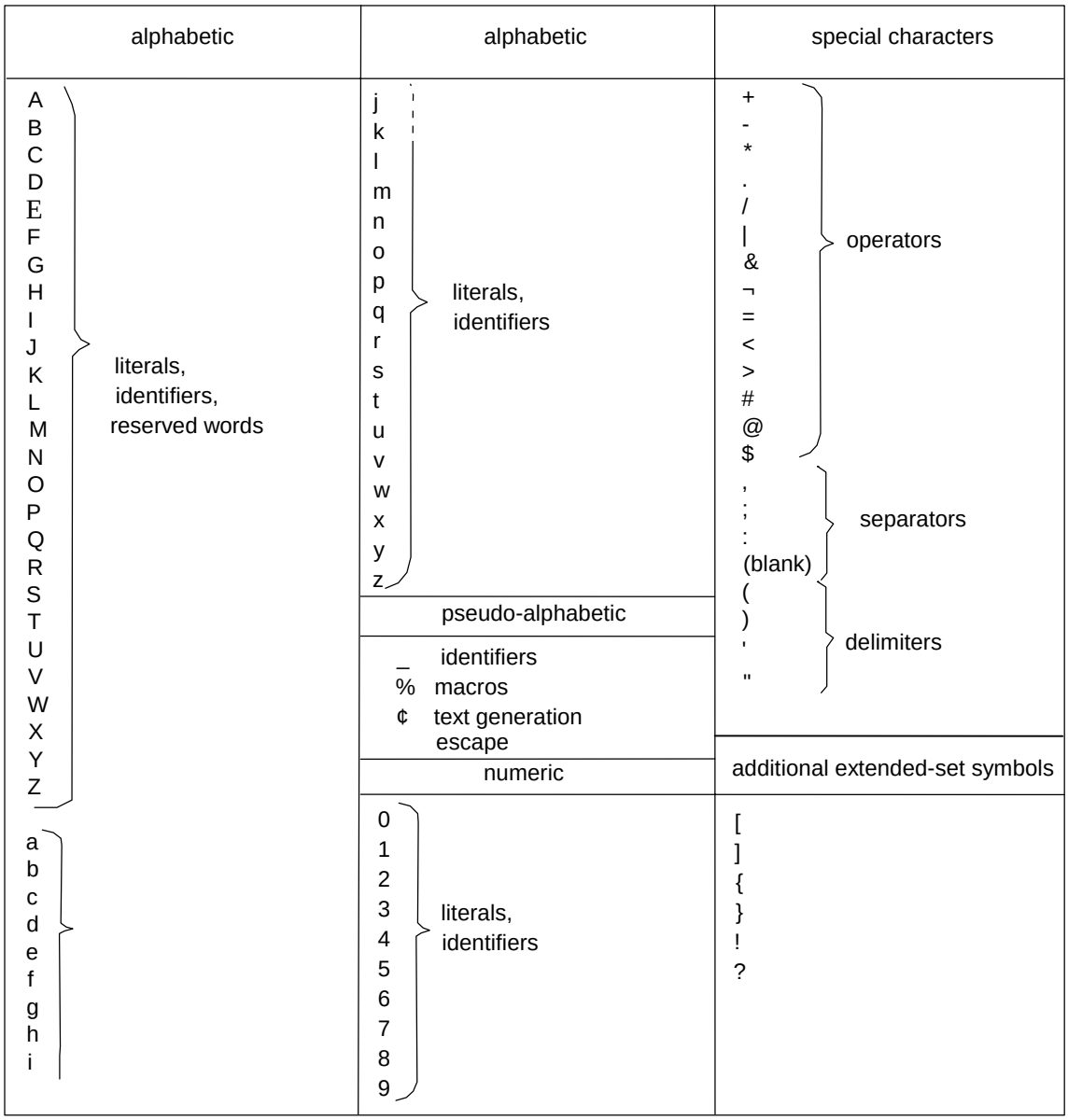


Figure 2-2 HAL/S character set

2.3 HAL/S Primitives.

HAL/S syntax diagrams ultimately express all syntactical elements in terms of a small number of special characters and pre-defined primitives. Primitives are constructed from the characters comprising the HAL/S restricted character set. There are three broad classes of primitives: “reserved words”, “identifiers”, and “literals”.

2.3.1 Reserved Words.

As their names suggest, reserved words are names recognized to have standard meanings within the language and which are unavailable for any other use. With the exception of %macro names, they are constructed from alphabetic characters alone. Reserved words fall into three categories: keywords, %macro, and built-in function names. In the syntax diagrams, and in the accompanying text, reserved words are indicated by upper case characters. A list of keywords is given in Appendix B, and of built-in function names in Appendix C.

2.3.2 Identifiers.

An identifier is a name assigned by the programmer to be a variable, label, or other entity. Before its attributes are defined, it is syntactically known as an <identifier>. Each valid <identifier> must satisfy the following rules:

- the total number of characters must not exceed 32;
- the first character must be alphabetic;
- any character except the first may be alphabetic or numeric;
- any character except the first or the last may be a “break character” (_).

The definition of an <identifier> generally establishes its attributes, and, in particular, its type. Thereafter, because its type is known, it is given one of the following names, as appropriate:

<label>
<process-event name>
< § var name>
<template name>

where § ≡

arith (arithmetic)
char (character)
bit
event
structure

The manner in which its attributes are established is discussed in Section 4. The manner in which it is thereafter referenced is discussed in Section 5.

2.3.3 Numbers.

HAL/S supports two numeric types: INTEGER and SCALAR.

INTEGER type provides all the signed integers in some finite range. INTEGER DOUBLE supports a larger range than INTEGER single.

SCALAR type is represented as floating point numbers. As such they are an approximation to the signed reals (engineering numbers) within some finite range and with some finite precision. SCALAR DOUBLE supports a larger range and/or greater precision than SCALAR single.

2.3.4 Literals.

Literals are groups of characters expressing their own values. During the execution of a body of HAL/S code their values remain constant. Differing rules apply for the formation of literals of differing type.

RULES FOR ARITHMETIC LITERALS:

1. No distinction is made between integer and scalar valued literals. They take on either integer or scalar type according to their context. Similarly, no distinction is made between single and double precision. Consequently, arithmetic literals can be represented by the single syntactical form <number>.
2. The generic form of a <number> is:

   ±dddddd.dddddddd<exponents>

   where d = decimal digit.Any number of decimal digits to an implementation dependent maximum, including none, may appear before or after the decimal point. The sign and the decimal point are both optional. Any number of <exponents> to an implementation dependent maximum may optionally follow.
3. The form of the <exponents> may be:

   B<power> = $2^{<power>}$
   E<power> = $10^{<power>}$
   H<power> = $16^{<power>}$

   where <power> is a signed integer number. The valid range of values of <power> is implementation dependent.

examples:
0.123E16B-3
45.9
-4

RULES FOR BIT LITERALS:

1. Literals of bit type are denoted syntactically by <bit literal>.
2. They have one of the forms shown below:

BIN <repetition>	‘bbbbbbb’	where	b =	binary digit
OCT <repetition>	‘ooooooo’		o =	octal digit
HEX <repetition>	‘hhhhhhh’		h =	hexadecimal digit
DEC	‘ddddddd’		d =	decimal digit

   The <repetition> is optional and consists of a parenthesized positive integer number. It indicates how many times the following string is to be used in creating the value. The number of digits lies between 1 and an implementation dependent maximum.
3. The following abbreviated forms are allowed:

```
TRUE ≡ ON ≡ BIN ‘1’
FALSE ≡ OFF ≡ BIN ‘0’
```

examples:

```
BIN‘11011000110’
HEX(3)‘F’
```

RULES FOR CHARACTER LITERALS:

1. Literals of character type are denoted syntactically by <char literal>.
2. They have one of the two following forms:

   'ccccccc'

   CHAR <repetition> 'ccccccc'

   Where c is any character in the HAL/S extended character set. The <repetition> consists of a parenthesized positive integer literal. It indicates how many times the following string is to be used in creating the value. The number of characters lies between zero and an implementation dependent maximum.
3. A null character literal (zero characters long) is denoted by two adjacent apostrophes.
4. Since an apostrophe delimits the string of characters inside the literal, an apostrophe must be represented by two adjacent apostrophes; i.e., the representation of "dog's" would be 'DOG' 'S'.
5. Within a character literal, a special "escape" mechanism may be employed to indicate a character other than one in the HAL/S extended character set. "¢" is defined to be the "escape" character within this context. In accordance with an implementation dependent mapping scheme, HAL/S characters will be assigned alternate character values. Inclusion of these alternate values in a string literal is achieved by preceding the appropriate HAL/S character by the proper number of "escape" characters. The specified character with the "escape" character(s) preceding it will be interpreted as a single character whose value is defined by the implementation.

   Since "¢" is used as the "escape" character, specification of the character "¢" as a literal itself must be done via the alternate character mechanism, i.e., an implementation will designate an alternate value for some HAL/S character to be the character "¢".

examples:

```
' '
'ONE TWO THREE'
'DOG"S'
'AB¢AD'     ]
'AB¢¢AD'    ]   The implication that ¢A and ¢¢A have been defined as alternate characters.
```

2.4 One- and Two-Dimensional Source Formats.

In preparing HAL/S source text, either single or multiple line format may be used. In the single line or "1-dimensional" format, exponents and subscripts are written on the same line as the operands to which they refer. In the multiple line or "2-dimensional" format, exponents are written above the line containing the operands to which they refer, and subscripts are written below it. Of the two formats, the 2-dimensional is considered standard since it closely parallels usual mathematical practice.

RULES FOR EXPONENTS:

1. In the syntax diagrams, the 1-dimensional format is assumed for clarity. The operation of taking an exponent is denoted by the operator **.

   examples:
   $A^J \rightarrow A^{**}J$
   $A^{JK} \rightarrow A^{**}J^{**}K$

2. Operations are evaluated right to left (see Section 6.1.1).
3. If an exponent is subscripted, the subscript must be written in the 1-dimensional format.

RULES FOR SUBSCRIPTS:

1. In the syntax diagrams, 2-dimensional format is assumed for clarity. Two special symbols are used to denote the descent to a subscript line, and the return from it:

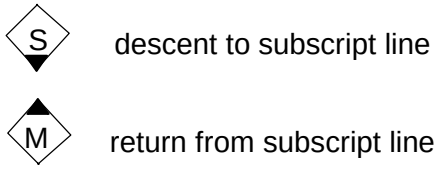


Figure 2-3 Subscript Diagram

   Effectively they delimit the beginning and end of a subscript expression, respectively.

2. The 1-dimensional format of a subscript expression consists of delimiting it at the beginning by $(and at the end by a right parenthesis.

   examples:
   $A_{K+2} \rightarrow$ `A$(K+2)`

3. For certain simple forms of subscript, the parentheses may be omitted. These forms are:
   - a single <number>
   - a single <arith var name> (see Section 5).

   examples:
   $A \rightarrow$ `A$J`

4. If a subscript expression contains an exponentiation operation, the latter must be written in the 1-dimensional format.

2.5 Comments and Blanks in the Source Text.

Any HAL/S source text consists of sequences of HAL/S primitives interspersed with special characters. It is obviously of great importance for a compiler to be able to tell the end of one text element from the beginning of the next. In many cases the rules for the formation of the primitives are sufficient to define the boundary. In others, a blank character is required as a separator. Blanks are legal in the following situations:

between two primitives;

between two special characters;

between a primitive and a special character.

Blanks are necessary (not just legal) between two primitives. With respect to string (bit and character) literals, the single quote mark serves as a legal separator.

Comments may be imbedded within HAL/S source text wherever blanks are legal. A comment is delimited at the start by the character pair /*, and at the end by the character pair */. Any characters in the extended character set may appear in the comment (except, of course, for * followed by /). There are implementation dependent restrictions on the overflow on imbedded comments from line to line of the source text.

3.0 HAL/S BLOCK STRUCTURE AND ORGANIZATION

The largest syntactical unit in the HAL/S language is the “unit of compilation”. In any implementation, the HAL/S compiler accepts “source modules” for translation, and emits “object modules” as a result. Each source module consists of one unit of compilation, plus compiler directives for its translation.

At run time, an arbitrary number of object modules are combined to form an executable “program complex” [2]. Generally, a program complex contains three different types of object modules:

- program modules - characterized by being independently executable.
- external procedure and function modules - characterized by being callable from other modules.
- compool modules - forming common data pools for the program complex.

Each module originates from a unit of compilation of corresponding type.

3.1 The Unit of Compilation.

Each unit of compilation consists of a single PROGRAM, PROCEDURE, FUNCTION, or COMPOOL block of code, possibly preceded by one or more block templates. Templates, in effect, provide the code block with information about other code blocks with which it will be combined in object module form at run time.

SYNTAX:

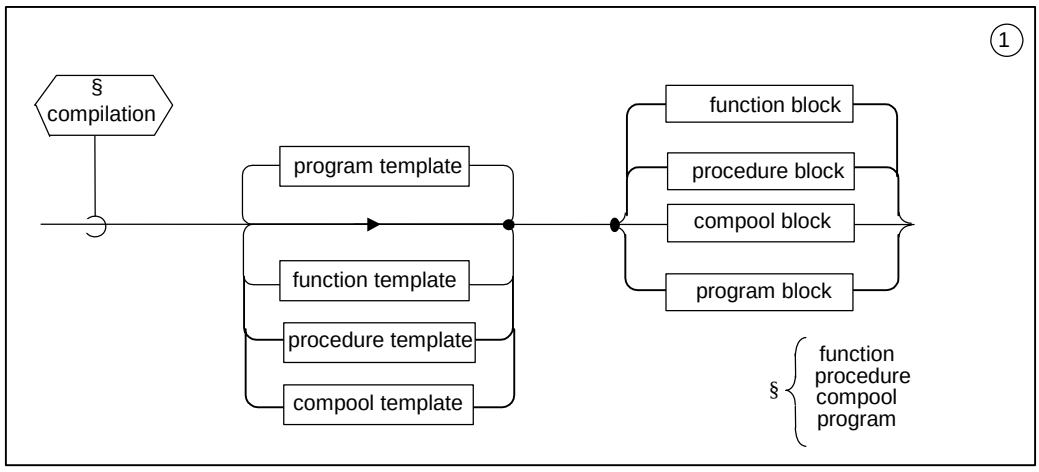


Figure 3-1 unit of compilation - #1

SEMANTIC RULES:

1. A program <compilation> is one containing a <program block>. Its object module in the program complex may be activated by the Real Time Executive (see Section 8), or by other means dependent on the operating system. The <program block> is described in Section 3.2.

2. A program complex is executable within the framework of an executive operating system, and a run time utility library.

2. A procedure or function <compilation> is one containing a <procedure block> or <function block>, respectively. Its object module in the program complex is executed by being invoked by other program, procedure, or function modules. Both <procedure block>s and <function block>s are described in Section 3.3.
3. A compool <compilation> is one containing a <compool block> specifying a common data pool potentially available to any program, procedure, or function module in the program complex. The <compool block> is described in Section 3.5.
4. The code block in any <compilation> except a compool <compilation> may contain references to data in a compool <compilation>, references to other <program block>s, and invocations of external <procedure block>s or <function block>s in other <compilation>s. A <compilation> making such references must precede its code block with a block template for each <program block>, <procedure block>, <function block>, or <compool block> referenced. Block templates are described in Section 3.6.

3.2 The PROGRAM Block.

The PROGRAM block delimits a main, independently executable body of HAL/S code.

SYNTAX:

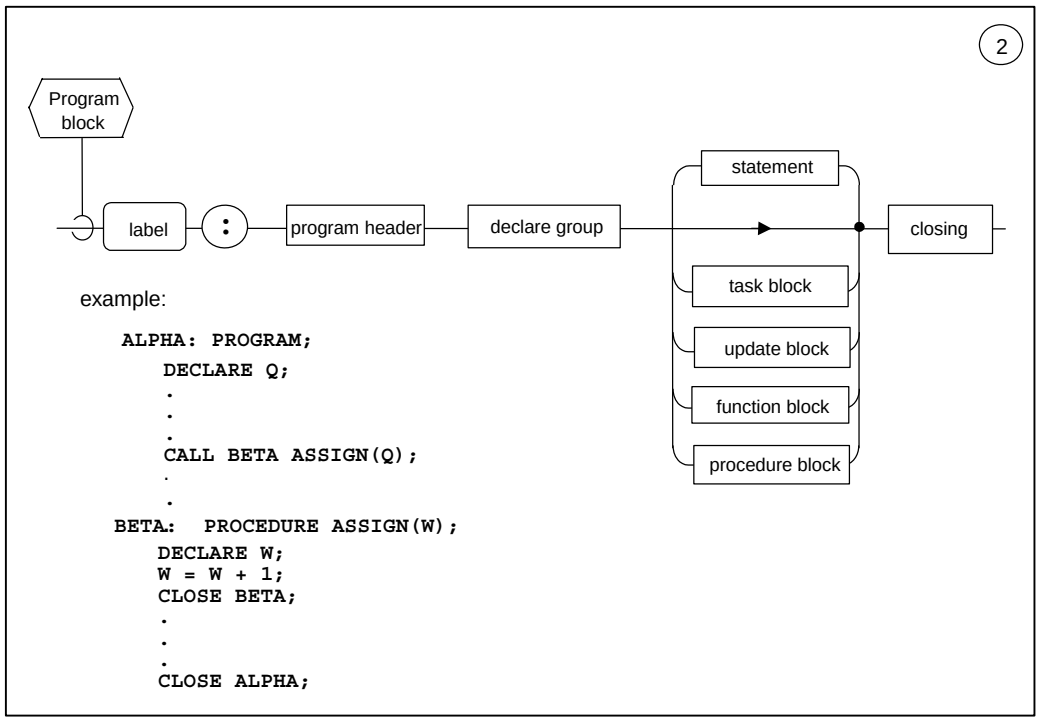


Figure 3-2 PROGRAM block - #2

SEMANTIC RULES:

1. The name of the <program block> is given by the <label> prefacing the block.
2. The <program block> is delimited by the <program header> statement at beginning, and a <closing> at the end. These two delimiting statements are described in Sections 3.7.1 and 3.7.4, respectively.
3. The contents of a <program block> consists of a <declare group> used to define data local to the <program block>, followed by any number of executable <statement>s.
4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement> may modify this normal sequencing in a well-defined way.
5. PROCEDURE, FUNCTION, TASK, and UPDATE blocks may appear nested within a <program block>. The blocks may be interspersed between the <statement>s of the <program block>, and with the exception of the UPDATE block are not executed in-line.
6. Execution of a <program block> is accomplished by scheduling it as a process under control of the Real Time Executive (see Section 8).

3.3 PROCEDURE, FUNCTION, and TASK Blocks.

PROCEDURE, FUNCTION, and TASK blocks share a common purpose in serving to structure HAL/S code into an interlocking modular form. The major semantic distinction between the three types of block is the manner of their invocation.

SYNTAX:

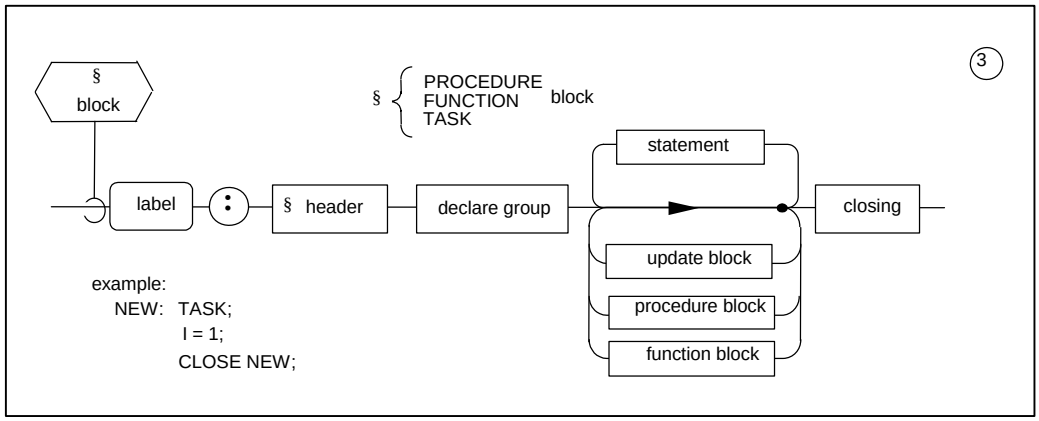


Figure 3-3 PROCEDURE, FUNCTION, TASK block - #3

SEMANTIC RULES

1. The name of the block is given by the <label> prefacing the block. The definition of a block label is considered to be in the scope of the outer block containing the block in question. Block names must be unique within any compilation unit.
2. The block is delimited at its beginning by a header statement characteristic of the type of block, and at the end by a <closing>. The delimiting statements are described in Sections 3.7.1 through 3.7.4.
3. The contents of a block consists of a <declare group> used to declare data local to

the block, followed by any number of executable <statement>s.

4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement> may modify this normal sequence in a well-defined way.
5. The block may contain further nested PROCEDURE, FUNCTION, and UPDATE blocks. An UPDATE block may not appear within an UPDATE block at any level of nesting. The nested blocks may appear interspersed between the <statement>s of the outer block, and except for the UPDATE block are not executed in-line. A consequence of this rule is that PROCEDURE and FUNCTION blocks may be nested within each other to an arbitrary depth.
6. The execution of a <task block> is invoked by scheduling it as a process under the control of the Real Time Executive (see Section 8). Execution of a <procedure block> is invoked by the CALL statement (see Section 7.4). Execution of a <function block> is invoked by the appearance of its name in an expression (see Section 6.4).
7. A <procedure block> or <function block> may result in either a single out-of-line expansion or an in-line expansion at each invocation. The semantics of a block invocation is independent of the way it is expanded.
8. A <task block> may not appear within a DO...END group.
9. In the <declare group> of a PROCEDURE or FUNCTION block which forms the outermost code block of a <compilation unit>, some implementations may require all formal parameters to be declared before any local data.

3.4 The UPDATE Block.

The UPDATE block is used to control the sharing of data by two or more real time processes. Its functional characteristics in this respect are described in Section 8.

SYNTAX:

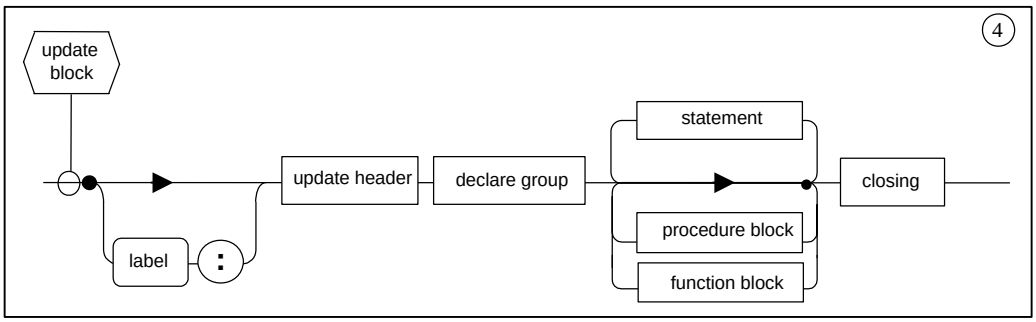


Figure 3-4 UPDATE block - #4

SEMANTIC RULES:

1. If present, the <label> prefacing the <update block> gives the name of the block. If <label> is absent, the <update block> is unnamed.

2. The block is delimited at its beginning by an <update header> statement, and at the end by a <closing>. The delimiting statements are described in Sections 3.7.1 and 3.7.4.
3. The contents of the block consist of a <declare group> used to declare data local to the <update block>, followed by any number of executable <statement>s.
4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement>s may modify this normal sequencing in a well defined way.
5. Only PROCEDURE and FUNCTION blocks may be nested within an <update block>. The nested blocks may appear interspersed between the <statement>s of the block, and are not executed in-line.
6. An <update block> is treated like a <statement> in that it is executed in-line. In this respect it is different from other code blocks.
7. The following <statement>s are expressly forbidden inside an <update block> in view of its special protective function:
   - I/O statements (see Section 10);
   - invocations of <procedure block>s or <function block>s not themselves nested within the <update block>;
   - real-time programming statements, except for the SIGNAL, SET, and RESET statements (see Section 8.8).

3.5 The COMPOOL Block.

The COMPOOL block specifies data in a common data pool to be shared at run time by a number of program, procedure, and function modules.

SYNTAX:

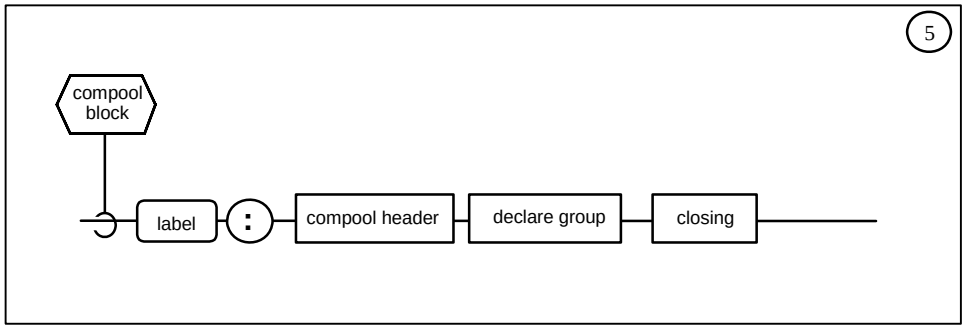


Figure 3-5 COMPOOL block - #5

SEMANTIC RULES:

1. The name of the block is given by the <label> prefacing the block.
2. The block is delimited at its beginning by a <compool header> statement, and at its end by a <closing>. The delimiting statements are described in Sections 3.7.1 and 3.7.4.
3. The contents of the block consist merely of a <declare group> used to define the data constituting the compool. In no sense is a <compool block> to be regarded as an executable body of code.

4. The maximum number of <compool block>s existing in a program complex is implementation dependent.

3.6 Block Templates.

In a <compilation>, block templates are used to provide the outermost code block of the <compilation> with information concerning external code or data blocks. Depending upon the implementation, the translation of program, procedure, function, and compool <compilation>s may automatically generate the corresponding block templates, to be included in other <compilation>s by compiler directive.

There are four kinds of block templates, PROGRAM, PROCEDURE, FUNCTION, and COMPOOL templates, all being syntactically similar (see Section 3.1).

SYNTAX:

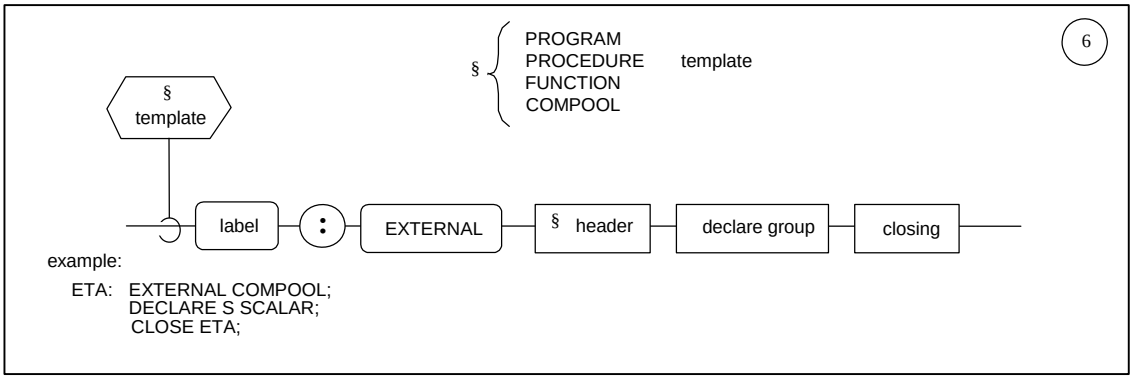


Figure 3-6 PROGRAM, PROCEDURE, FUNCTION, COMPOOL template - #6

SEMANTIC RULES

1. The <label> of the template constitutes the template name. It is the same name as that of the code block to which the template corresponds.
2. The block template is delimited at its beginning by a header statement identical with the header statement of the corresponding code block, and at the end by a <closing>. The delimiting statements are described in Sections 3.7.1 and 3.7.4.
3. The contents of the block template consist only of a <declare group>, which has the following significance:
   - in a program <template>, the <declare group> contains no statements. All information about external programs is contained in the <program header>;
   - in a <compool template>, the <declare group> is used to declare a common data pool identical with that of the corresponding <compool block>;
   - in a <procedure template> or <function template>, the <declare group> is used to declare the formal parameters of the corresponding <procedure block> or <function block> (see Sections 3.7.2 and 3.7.3).
4. The keyword EXTERNAL preceding the header statement of the block template distinguishes it from an otherwise identical code block. To a HAL/S compiler the keyword is in effect a signal to prevent the compiler from generating object code for the block and setting aside space for the data declared.

3.7 Block Delimiting Statements.

Both code blocks and block templates are delimited at the beginning by a header statement characteristic of their type, and at the end by a <closing> statement. In all code blocks except for the COMPOOL block, the header statement is the first statement of the block to be executed upon entry. A COMPOOL block, containing only declarations of data, is, of course, not executable at all.

3.7.1 Simple Header Statements

Simple header statements are those which specify no parameters to be passed into or out of the block. They are the compool, program, task, and update header statements.

SYNTAX:

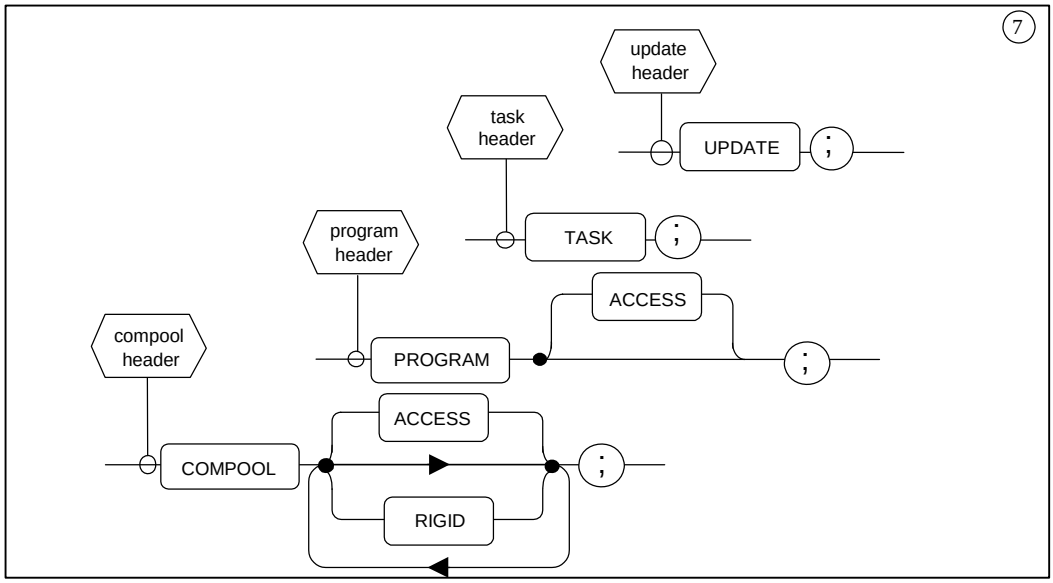


Figure 3-7 COMPOOL, PROGRAM, TASK, UPDATE header statement - #7

SEMANTIC RULES:

1. The type of the code block or template is determined by the type of the header statement, which is in turn indicated by one of the keywords COMPOOL, PROGRAM, TASK, and UPDATE.
2. The keyword ACCESS causes managerial restrictions to be placed upon the usage of the block in question. The manner of enforcement of the restriction is implementation dependent.
3. The keyword RIGID causes COMPOOL data (except for data with the REMOTE attribute) to be organized in the order declared and not rearranged by the compiler.

3.7.2 The Procedure Header Statement.

The procedure header statement delimits the start of a <procedure block> or <procedure template>.

SYNTAX:

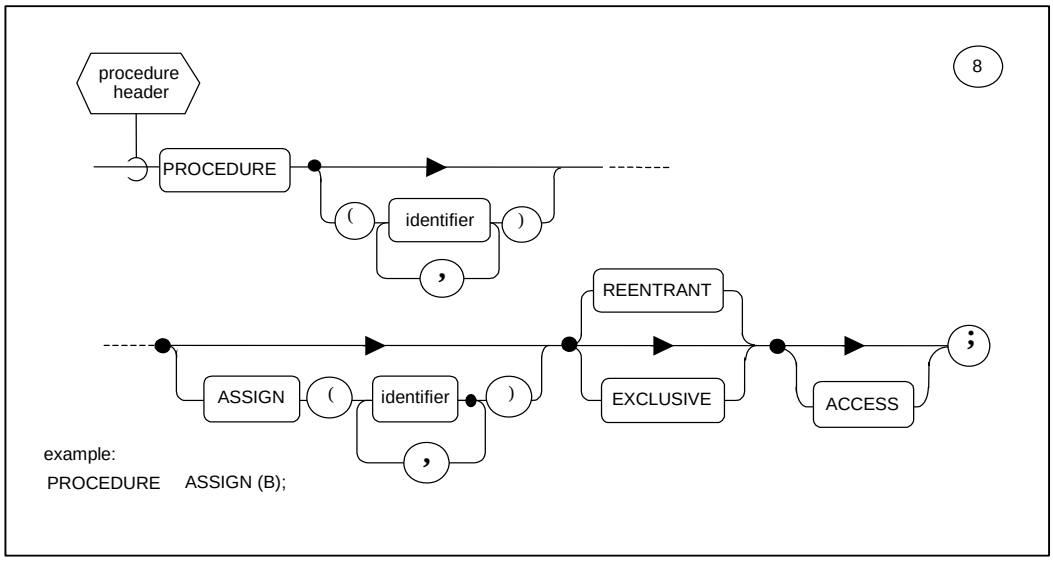


Figure 3-8 PROCEDURE header statement - #8

SEMANTIC RULES:

1. The keyword PROCEDURE identifies the start of a <procedure block> or <procedure template>. It is optionally followed by lists of “formal parameters” which correspond to “arguments” in the invocation of the procedure by a CALL statement (see Section 7.4).
2. The <identifier>s in the list following the PROCEDURE keyword are called “input parameters” because they may not appear in any context inside the code block which may cause their values to be changed.
3. The <identifier>s in the list following the ASSIGN keyword are called “assign parameters” because they may appear in contexts inside the code block in which new values may be assigned to them. They may, of course, also appear in the same contexts as input parameters.
4. Data declarations for all formal parameters must appear in the <declare group> of the <procedure block> or <procedure template>.
5. If the <procedure header> statement specifies neither of the keywords REENTRANT or EXCLUSIVE, then only one real time process (see Section 8) may be executing the <procedure block> at any one time; however, there is no enforcing protective mechanism. If the keyword EXCLUSIVE is specified, then such a protective mechanism does exist. If an EXCLUSIVE <procedure block> is already being executed by a real time process when a second process tries to invoke it, the second process is forced into the stall state (see Section 8) until the first has finished executing it. If the keyword REENTRANT is specified, then two or more processes may execute the <procedure block> “simultaneously”.

6. The keyword REENTRANT indicates to the compiler that reentrancy is desired. However, other attributes and conditions may conflict with this overall objective. The following effects should be noted:
   - STATIC data is allocated statically and initialized statically. There is only one copy of static data which must be shared by all processes simultaneously executing the block. Hence, in coding REENTRANT blocks care must be taken not to assume that STATIC variables participate in the reentrancy.
   - AUTOMATIC data is allocated dynamically and initialized dynamically. Every process simultaneously executing the block gets its own initialized copy of the data on entry into the block. In general, all local data in a REENTRANT block should be declared with the AUTOMATIC attribute.
   - Procedures and functions defined within a REENTRANT block must also possess the REENTRANT attribute if they, too, declare local data which is required to participate in the reentrancy.

   In addition, for reentrancy to be preserved, the following rules must be observed:
   - Update blocks[3] and inline functions within a REENTRANT block may not declare any local data, STATIC or AUTOMATIC, because the update block does not inherit the reentrant attribute from the enclosing procedure declaration;
   - A procedure or function called by a REENTRANT block must itself also be REENTRANT.
7. The keyword ACCESS may be attached to the <procedure header> of a <procedure template> and its corresponding external <procedure block>. It denotes that managerial restrictions are to be placed on which <compilation>s may reference the <procedure block>. The manner of enforcement is implementation dependent.

3.7.3 The Function Header Statement.

The function header statement delimits the start of a <function block> or <function template>.

SYNTAX:

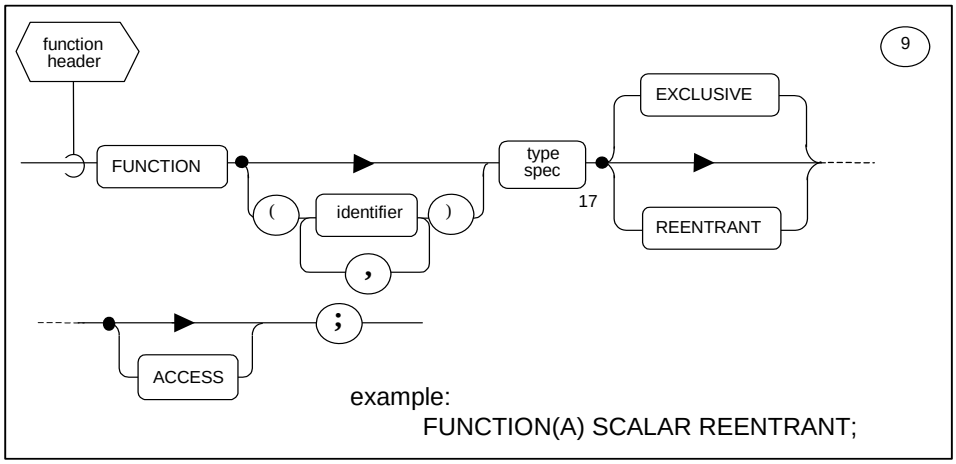


Figure 3-9 FUNCTION header statement - #9

3. Any use of update blocks and LOCK data, or of EXCLUSIVE procedure or function blocks should be carefully analyzed with respect to unfavorable timing problems if a procedure is reentered by a higher priority process.

SEMANTIC RULES:

1. The keyword FUNCTION identifies the start of a <function block> or <function template>. It is optionally followed by a list of “formal parameters” which are substituted by corresponding “arguments” in the invocation of the <function block> (see Section 6.1.1).
2. The <identifier>s in the list following the FUNCTION keyword are “input parameters” since they may not appear in any context inside the <function block> which may cause their values to be changed.
3. Data declarations for all the formal parameters must appear in the <declare group> of the <function block> or the <function template>.
4. <type spec> identifies the type of the <function block> or <function template>. A <function block> may be of any type except EVENT. The formal description of the type specification given by <type spec> is given in Section 4.7.
5. If the <function header> statement specifies neither of the keywords REENTRANT or EXCLUSIVE, then only one real time process (see Section 8) may be executing the <function block> at any one time; however, there is no enforcing protective mechanism. If the keyword EXCLUSIVE is specified, then such a protective mechanism does exist. If an EXCLUSIVE <function block> is already being executed by a real time process when a second process tries to invoke it, the second process is forced into the stall state (see Section 8) until the first has finished executing it. If the keyword REENTRANT is specified, then two or more processes may execute the <function block> “simultaneously”.
6. The keyword REENTRANT indicates to the compiler that reentrancy is desired. However, other attributes and conditions may conflict with this overall objective. The following effects should be noted:
   - STATIC data is allocated statically and initialized statically. There is only one copy of STATIC data which must be shared by all processes simultaneously executing the block. Hence, in coding REENTRANTblocks, care must be taken not to assume that STATIC variables participate in the reentrancy.
   - AUTOMATICdata is allocated dynamically and initialized dynamically. Every process simultaneously executing the block gets its own initialized copy of the data on entry into the block. In general, all local data in a REENTRANTblock should be declared with the AUTOMATIC attribute.
   - Procedures and functions defined within a REENTRANT block must also possess the REENTRANT attribute if they, too, declare local data which is required to participate in the reentrancy.

   In addition, for reentrancy to be preserved, the following rules must be observed:
   - Update blocks[4] and inline functions within a REENTRANTblock may not declare any local data, STATIC or AUTOMATIC, because the update block does not inherit the reentrant attribute from the enclosing function declaration;
   - A procedure or function called by a REENTRANT block must itself also be REENTRANT.

4. Any use of update blocks and LOCK data, or of EXCLUSIVE procedure or function blocks should be carefully analyzed with respect to unfavorable timing problems if a function is reentered by a higher priority process.

7. The keyword ACCESS may be attached to the <function header> of a <function template> and its corresponding external <function block>. It denotes that managerial restrictions are to be placed on which <compilation>s may reference the <function block>. The manner of enforcement is implementation dependent.

3.7.4 The CLOSE Statement.

For all code blocks, COMPOOL blocks, and block templates, the CLOSE statement is the <closing> delimiter of the block.

SYNTAX:

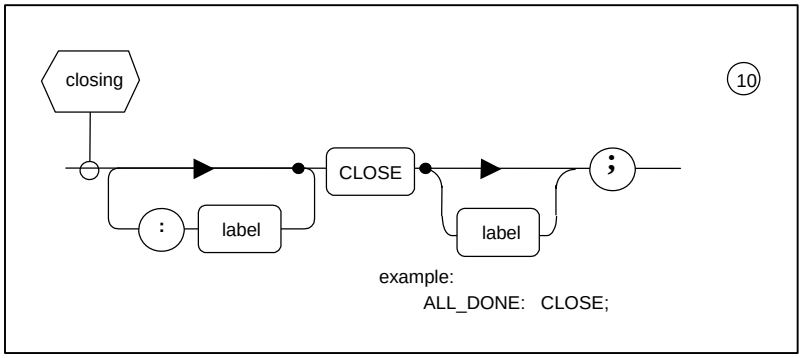


Figure 3-10 closing of block - #10

SEMANTIC RULES:

1. The <closing> of a code block or block template is denoted by the CLOSE keyword followed by an optional <label>. If present, <label> must be the name of the block.
2. Execution of the CLOSE statement causes a normal exit from a PROGRAM, PROCEDURE, TASK, or UPDATE block, and a run time error from a FUNCTION block. Exit from a function block must be achieved via the RETURN statement (see Section 7.5).
3. The <closing> of a COMPOOL or PROGRAM block template or a PROGRAM, COMPOOL, PROCEDURE, FUNCTION, TASK, or UPDATE block may be labeled as if it were a <statement>. The <closing>s of PROCEDURE and FUNCTION block templates cannot be labeled.

3.8 Name Scope Rules.

By using the code blocks described, and by taking advantage of their nesting property, the modularization of HAL/S <compilation>s may be achieved. An important consequence of the nesting property is the need to determine the “name scope” over which names defined in a code block are potentially known. Names (i.e., <identifier>s) to which name scope rules apply are generally either labels or variable names.

GENERAL RULES:

1. The name scope of a code block encompasses the entire contents of a block, including all blocks nested within it.
2. The name defined in a name-scope is known, and therefore able to be referenced, throughout that name-scope, including all nested blocks not redefining it. A name defined in a name-scope is not known outside that name-scope.
3. Names defined in all common data pools used by a <compilation> are considered to be defined in one name-scope which encloses the outermost code block of the <compilation>.

QUALIFICATIONS:

1. The name of a code block is taken to be defined in the name-scope immediately enclosing the block. A PROCEDURE or FUNCTION defined at the outermost level of compilation can be invoked from anywhere within the compilation.
2. The <label> of a statement is effectively unknown in blocks contained in the name scope where the <label> is defined. This is because a code block cannot be branched out of by using a GO TO statement (see Section 7.7).
3. Block labels must be unique throughout a unit of compilation.
4. Under particular, limited circumstances described in Section 4.3, the names of structure template nodes and terminals need not be unique.

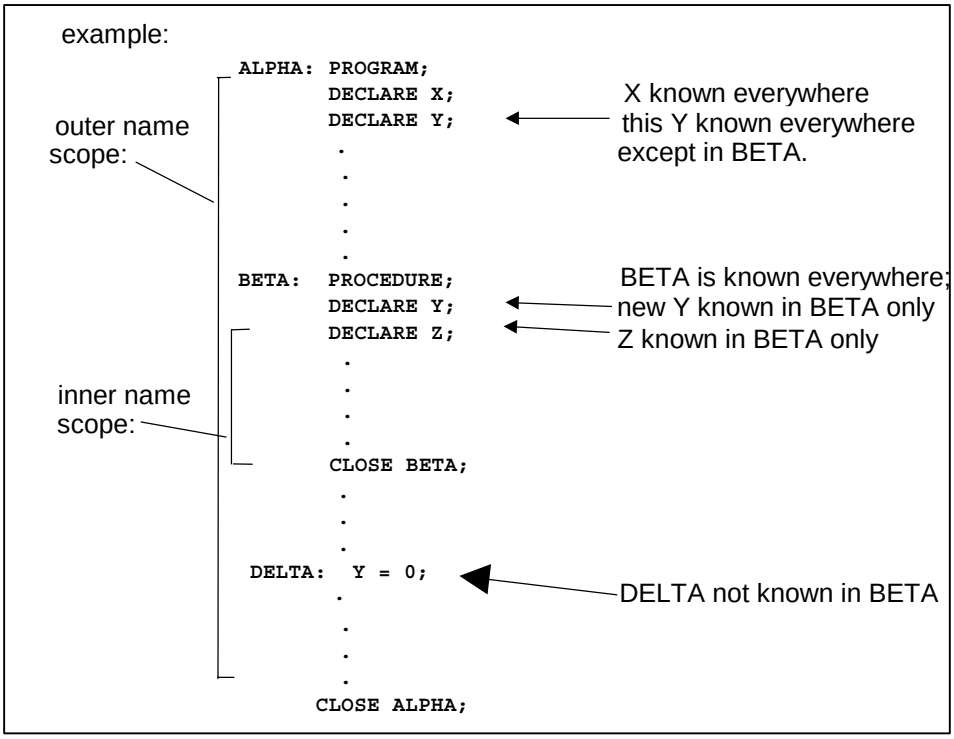


Figure 3-11 Name Scope Examples

4.0 DATA AND OTHER DECLARATIONS

The HAL/S language provides a comprehensive set of data types. To encourage clarity and decrease the frequency of errors of omission, all data is required to be declared in specific areas of a HAL/S compilation called “declare groups”. Occasionally the demands of a particular algorithm also require other kinds of declarations to be made. Figure 4-1 summarizes the relationship among the types and organizations.

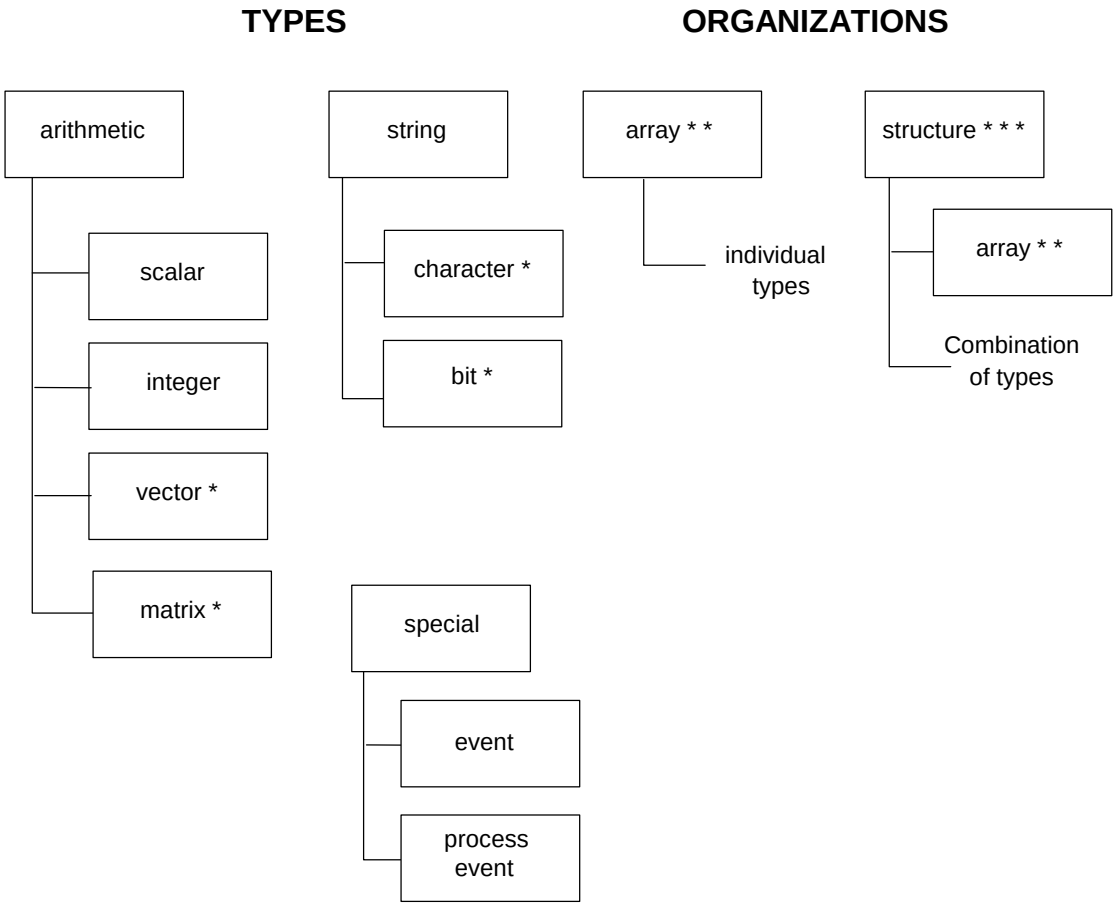


Figure 4-1 HAL/S DATA TYPES AND ORGANIZATIONS

* Component Subscripting (see Section 5.3.5) Allowed.

* * Array Subscripting Allowed.

* * * Structure Subscripting Allowed.

4.1 The Declare Group.

A <declare group> is a collection of data and other declarations. The position of <declare group>s within code blocks and block templates has been described in Section 3.

SYNTAX:

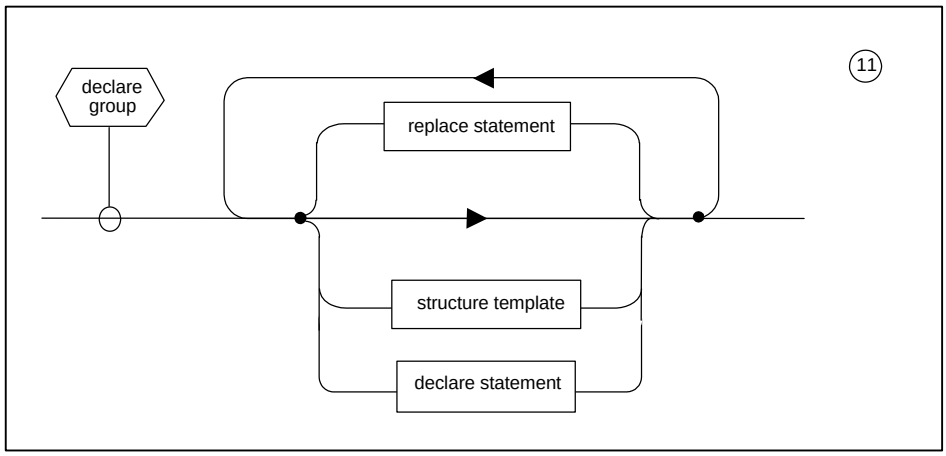


Figure 4-2 declare group - #11

SEMANTIC RULES:

1. A <declare group> may simply be empty, or it may contain <replace statement>s, <structure template>s, and <declare statement>s. The form of each of these constructs is defined in this Section.
2. The “name scope” (see Section 3.8) of <identifier>s defined in a <declare group> is the code block containing the <declare group> and potentially all code blocks nested within it.

4.2 The REPLACE Statement.

The REPLACE statement is used to define an identifier text substitution which is to take place wherever the identifier is referenced within the same name scope after its definition. The REPLACE statement constitutes a “source macro” definition.

4.2.1 Form of REPLACE Statement.

SYNTAX:

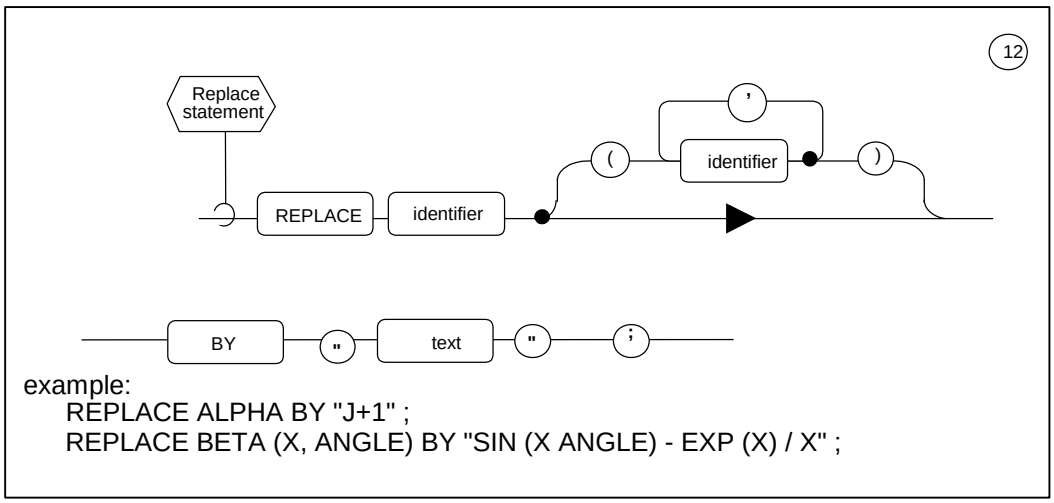


Figure 4-3 replace statement - #12

GENERAL SEMANTIC RULES:

1. The <identifier> following the keyword REPLACE is called the REPLACE name.
2. A REPLACE name may not appear as a formal parameter in a <procedure header> or <function header>.
3. A REPLACE name in an inner code block is never “replaced” as a result of another REPLACE statement located in an outer code block.
4. Nested replacement operations to some implementation dependent depth are allowed (i.e., the <text> of a <replacement statement> may contain a further <identifier> to be replaced).

SEMANTIC RULES: Simple Replacements

1. A simple replacement is a REPLACE statement with no parameter list following the <identifier>.
2. Whenever it is referenced, an <identifier> defined in a simple REPLACE statement is to be replaced by <text> of the definition as if <text> had been written directly instead of the source macro reference. Enclosing the reference within ¢ signs (e.g., ¢ALPHA¢) makes the <text> visible in the compiler listing.
3. <text> may consist of any HAL/S characters except instances of an unpaired double quote (“) character. A double quote character (”) is indicated within the <text> by two such characters in succession (“”).

SEMANTIC RULES: Parametric Replacements

1. A parametric replacement is defined by a REPLACE statement with a list of one or more parameters following the <identifier>. The maximum number of parameters allowed is an implementation dependent limit. Each parameter is itself a HAL/S <identifier>. It is known only locally to the REPLACE statement: its name may therefore be duplicated by names used for other <identifier>s in the name scope containing the REPLACE statement.
2. The <text> of a parametric REPLACE statement is composed of any HAL/S characters except instances of an unpaired double quote (“) character. A double quote character may be indicated within <text> by coding two such characters in succession. The <text> may contain, but is not required to contain, instances of the parameters of the REPLACE statement.

4.2.2 Referencing REPLACE Statements.

SYNTAX:

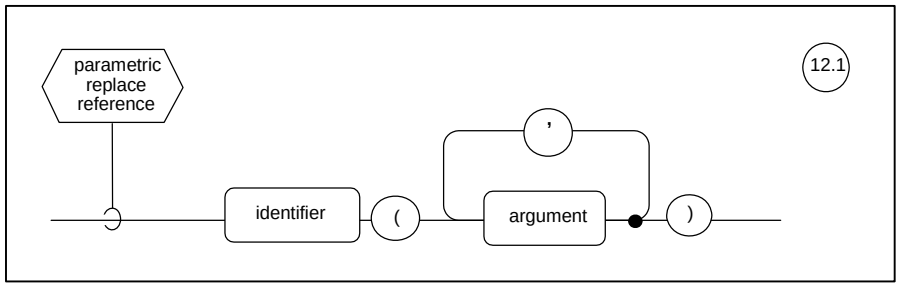


Figure 4-4 parametric replace reference - #12.1

SEMANTIC RULES:

1. A reference to a parametric REPLACE statement consists of the REPLACE name followed by a series of <argument>s enclosed in parentheses. The REPLACE name must have been defined previously within the name scope of the reference. The number of <argument>s must correspond to the number of parameters of the REPLACE statement being referenced. Enclosing the reference within ¢ signs (e.g., ¢CBETA(A,B)¢) makes the <text> visible in the compiler listing.
2. The <argument>s supplied in a parametric REPLACE reference are substituted for each occurrence of the corresponding parameter within the source macro definitions <text>. Note that if the parameter in question does not occur within the source macro definition's <text>, the <argument> is ineffective. <text> substitution is always completed before parsing.

Example:

```
REPLACE BETA(X,ANGLE) BY "SIN(X ANGLE) - EXP(X)/X";
       .
       .
       .
Z = BETA(Y,ALPHA);  WILL GENERATE SIN(Y ALPHA) - EXP(Y)/Y
```

3. In general, the <argument>s supplied in a parametric REPLACE reference comprise <text> separated by commas (subject to the specific exceptions listed below). As such, they conform to the preceding semantic rules for <text> with the following emendations:
   - Blanks are significant in <argument>s. Only the commas used to separate <argument>s are excluded from the <text> values substituted into the macro definition.
   - The <text> string comprising an <argument> may be empty. The value substituted in such a case is a null string.
   - Within each <argument> there must be an even number of apostrophe characters ('). The effect of this rule is to require that each character literal used must be completely contained within a single <argument>.
   - Within each <argument> there must be an even number of quotation mark characters ("). The effect of this rule is to require that the substitution of a nested REPLACE statement include the entire text of the replacement within a single <argument>.
   - Within each <argument> there must be a balanced number of left and right parentheses: for each opening left parenthesis there must be a corresponding right parenthesis.
   - Commas are not separators between <argument>s under the following circumstances:
     - within a character literal.
     - within REPLACE <text>.
     - nested within parentheses.

4.2.3 Identifier Generation

New identifiers may be generated by enclosing a reference to a simple REPLACE statement within ¢ signs. The effect is to make visible in the compiler listing, the catenation of the REPLACE <text> with the characters surrounding the construct. For example, REPLACE ABLE BY “BAKER”; then:

```
1. X = ¢ABLE¢YZ becomes X = BAKERYZ
2. CALL P_¢ABLE¢(Q,R,S); becomes CALL P_BAKER(Q,R,S);
```

¢ signs are taken in pairs, thus ¢X¢Y¢Z¢ is interpreted as ¢X¢Y¢Z¢.

4.2.4 Identifier Generation With Macro Parameters.

New identifiers may be generated for text substitution within a source macro text by enclosing references to macro parameters within ¢ signs. The effect is the compile-time catenation of the corresponding macro arguments with the characters surrounding the ¢-enclosed parameter (a blank is considered as a character). For example:

```
REPLACE ABLE(X,Y) BY
      “P = ¢X¢QRS+Y;
      CALL SUB_¢X¢; ”;
```

Then the reference ABLE (V, A) causes the following substitutions.

```
      P = VQRS+A;
      CALL SUB_V;
```

Figure 4-5 Creating Identifiers With Replace Macros

Enclosing the entire reference with ¢ signs, i.e., ¢ABLE(V, A)¢ makes the text with the new identifiers visible in the compiler listing (see Section 4.2.2).

4.3 The Structure Template.

In HAL/S, a “structure” is a hierarchical organization of generally nonhomogeneous data items. Conceptually, the form of the organization is a “tree”, with a “root”, “branches”, and with the data as “leaves”. The definition of the “tree organization” (the manner in which the root is connected to the branches, and branches to leaves) is separate from the declaration of a structure having that organization. The tree organization is defined by a <structure template> described below. The description of the declaration of structures is deferred to later subsections.

The following figure represents a typical tree organization.

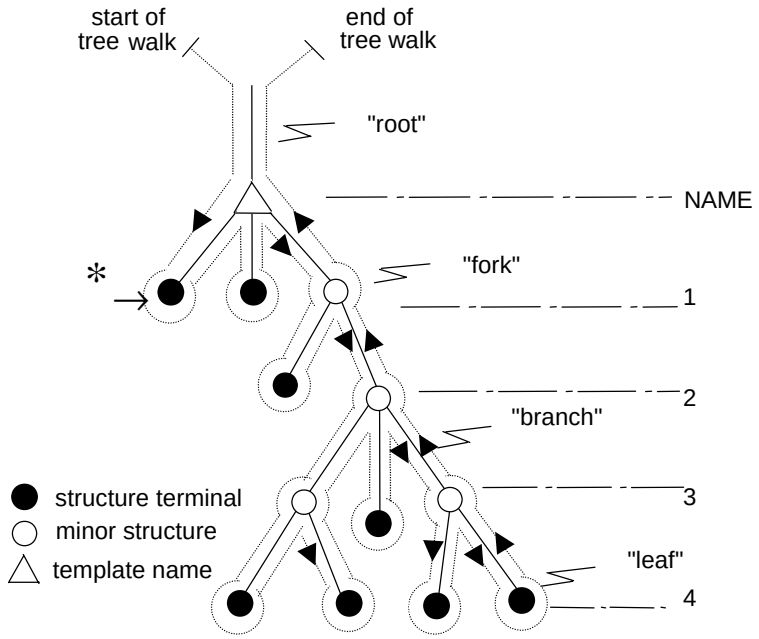


Figure 4-6 Tree diagram for a typical structure template

INTERPRETATIONS:

1. The “template name” is at the root of the tree organization.
2. The named “leaves” and “forks” in the branches are at numbered levels below the root. Leaves and forks are called “structure terminals” and “minor structures”, respectively.
3. The “tree walk” shown can provide an unambiguous linear description of the tree organization. The syntactical form of the <structure template> corresponding to a tree organization calls for the names of minor structures and structure terminals to be defined in the same order that the tree walk passes them on the left, as indicated by the arrow at * in the diagram.
4. The tree organization of two templates are considered to be equivalent for the purposes of various HAL/S statement contexts only if the tree forms are identical, and the type and attributes of all nodes in the tree agree. An implication of this rule becomes apparent: if two corresponding terminal nodes of otherwise equivalent structures reference different structure template names, then the structure templates containing these terminal nodes are <u>not</u> identical.

The syntactical form of a <structure template> is now given:
SYNTAX:

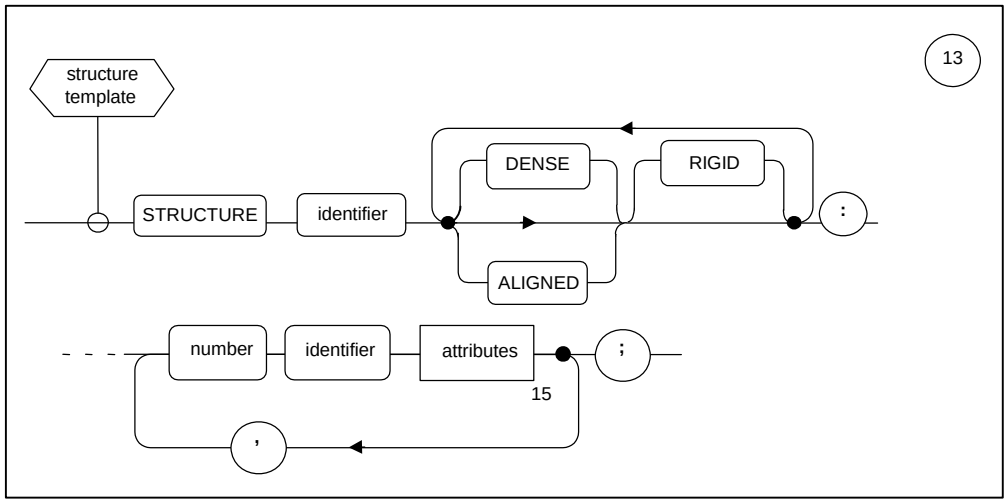


Figure 4-7 structure template statement - #13

GENERAL RULES:

1. The <template name> of the <structure template> is given by the <identifier> following the keyword STRUCTURE.
2. The operational keywords DENSE and ALIGNED denote data packing attributes to be applied to all <identifiers> declared with the <structure template>. At each level of a <structure template>, either the DENSE or ALIGNED packing attribute is in effect, subject to modification by use of DENSE and ALIGNED as minor <attributes>. The choice used in the <structure template> gives the default value for the whole template. This packing attribute is then inherited from higher to lower levels in the structure unless the <attributes> of a minor structure or terminal element modify the choice. Details of the allocation algorithm used for DENSE and ALIGNED data are implementation dependent.
3. The keyword RIGID causes data to be in the sequential order declared within the <structure template>. This attribute is then inherited from higher to lower levels in the structure. Details of the allocation algorithm used for RIGID are implementation dependent. Note that the absence of the keyword RIGID permits compiler reorganization of data.
4. In each definition, <number> is a positive integer specifying the level of the tree at which the definition is effective. Numbering is sequential starting with 1.
5. The level of definition in conjunction with the order of definition is sufficient to distinguish between a minor structure and a structure terminal.
6. In the form <identifier> <attributes>, <identifier> is the name of the minor structure or structure terminal defined. The applicable <attributes> are described in Section 4.5.
7. If the <attributes> specify a structure template <type spec> (see Section 4.7), then the template of the structure is being included as part of the template being defined.
8. The minor structures and structure terminals of the template (the forks and leaves) are

sequentially defined following the colon. The order of definition has already been described.

9. Each definition of a minor structure of structure terminal is separated from the next by a comma.

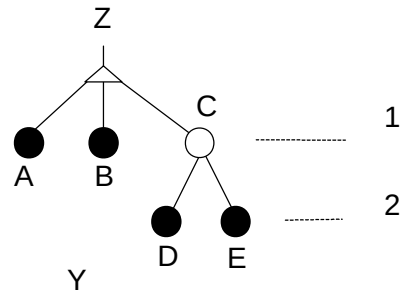
NAME UNIQUENESS RULES:

1. <template names> may duplicate <identifiers> of any other kind within a given name scope, but may not duplicate other <template names>.
2. In a given name scope, if a <template name> is used exclusively in qualified structure declarations, duplications of the <identifiers> used for nodes may occur under the following circumstances:
   - Any <identifier> used for a node in one template may duplicate an <identifier> used for a node in another template.
   - Any <identifier> used for a node in a given template may duplicate another <identifier> used for a different node in the same template, provided that a qualified reference can distinguish the two nodes.
3. In a given name scope, if a template is ever used for a non-qualified structure variable declaration, the duplications allowed under rule #2 within that template become illegal.

examples:

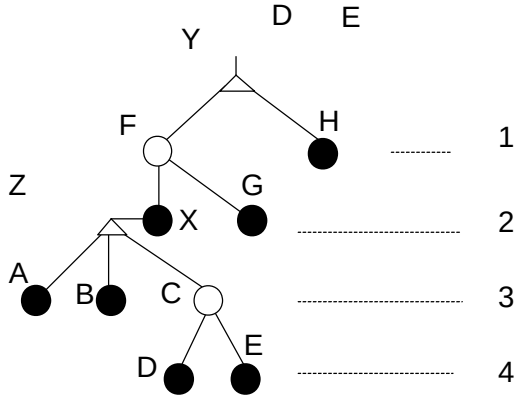
i) definition of a template z

```
STRUCTURE Z:
 1 A SCALAR,
 1 B VECTOR(4),
 1 C,
 2  D MATRIX(4,4),
   2  E BIT(3);
```



ii) definition of a template Y with Z nested within it

```
STRUCTURE Y:
 1 F,
    2 X Z-STRUCTURE,
    2 G INTEGER,
 1 H CHARACTER(10);
```



iii) equivalent form of template Y without nesting

```
STRUCTURE Y:
  1 F,
  2 X,
     3 A SCALAR,
     3 B VECTOR(4),
     3 C,
       4 D MATRIX(4,4),
       4 E BIT(3),
    2 G INTEGER,
   1 H CHARACTER(10);
```

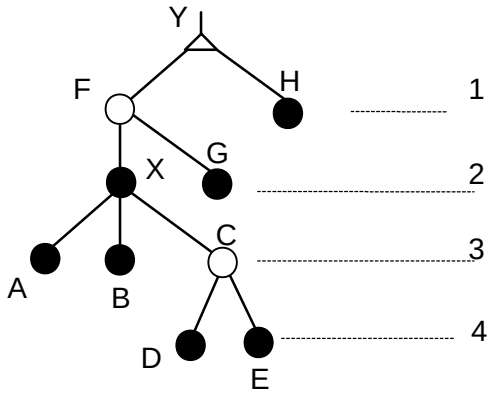


Figure 4-8 structure template examples

4.4 The DECLARE Statement.

The DECLARE statement is used to declare data names and labels, and to define their characteristics or <attributes>.

SYNTAX:

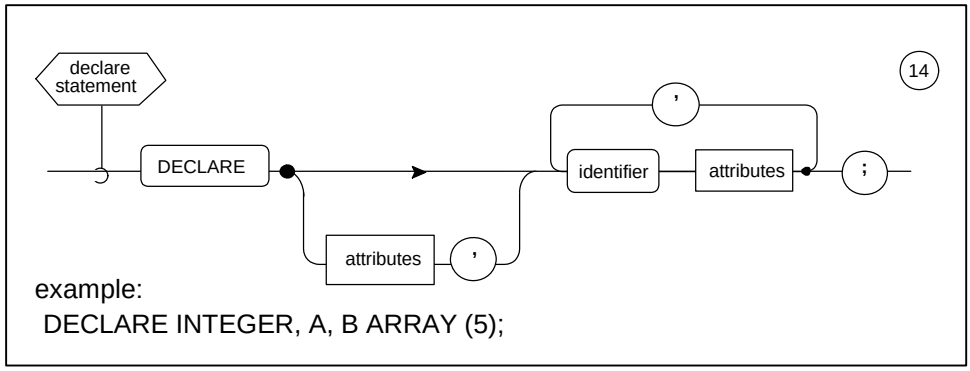


Figure 4-9 declaration statement - #14

SEMANTIC RULES:

1. Each <identifier> and its following <attributes> constitute the declaration of a data name or label. Each definition is separated from the next by a comma.
2. The generic characteristics, if any, of all <identifier>s to be declared are given by the "factored" <attributes> immediately following the keyword DECLARE. The <attributes> of a particular <identifier> must not conflict with the factored <attributes>.
3. The name scope of any of the <identifier>s defined in a <declare statement> is the code block containing the <declare group> of which the <declare statement> is a part (see Section 3.8). In any name scope all such <identifiers> must be unique.
4. There are two forms of <attributes>::: data declarative, and label declarative. The form determines whether an <identifier> is defined as a data name or a label.

4.5 Data Declarative Attributes.

Data declarative attributes are used to define an <identifier> to be a data name or part of a structure template, and to describe its characteristics. If <attributes> appear in a <declare statement>, the <identifier> defined is a "simple variable", or a "major structure" with a predefined template. If <attributes> appear in a <structure template>, the <identifier> defined is either a minor structure, or a structure terminal. Structure terminals have very similar properties to simple variables.

SYNTAX:

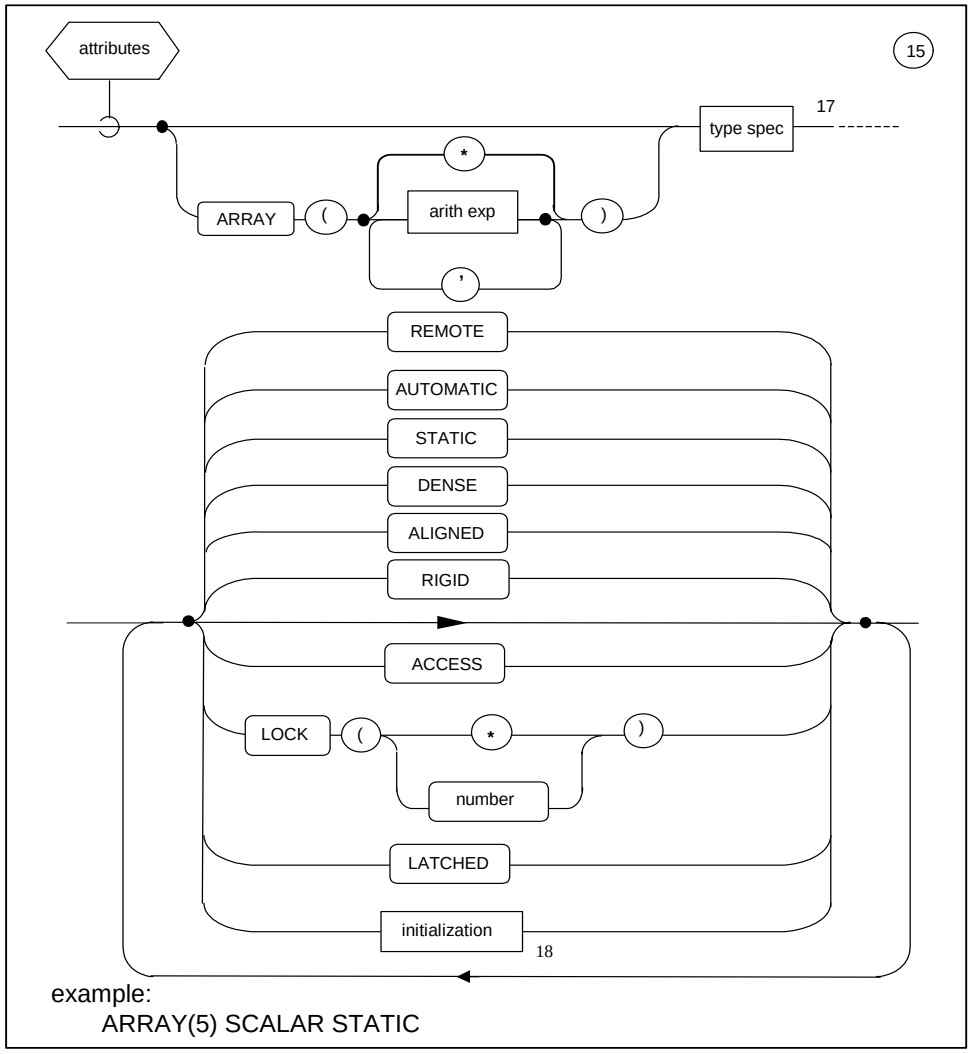


Figure 4-10 data declarative attributes - #15

GENERAL SEMANTIC RULES:

1. The <type spec> determines the type and possibly the precision of the <identifier> to which the <attributes> are attached. Type specifications are discussed in Section 4.7.
2. An optional array specification can precede the <type spec>. It starts with the keyword ARRAY; the following parenthesized list specifies the number of dimensions in the array, and the size of each dimension. The number N of <arith exp>s gives the number of dimensions of the array. <arith exp> is an unarrayed integer or scalar expression computable at compile time[5]. The value is rounded to the nearest integer, and indicates the number of elements in a dimension. Its value must lie between 2

5. See Appendix F.

and an implementation-dependent maximum. The maximum value for N is implementation dependent. A single asterisk denotes a linear array, the number of elements of which is greater than 1 but unknown at compile time.

3. Following the <type spec> a number of minor attributes applicable to the <identifier> can appear. These are:
   - STATIC/AUTOMATIC - the appearance of one of these keywords is mutually exclusive of the other. STATIC and AUTOMATIC refer to modes of initialization of an <identifier>, not to the allocation of its storage. The AUTOMATIC attribute causes an <identifier> with the <initialization> attribute to be initialized on every entry into the code block containing its declaration. The STATIC attribute causes such an <identifier> to be initialized only on the first entry into the code block. Thereafter its value on any exit from the code block is guaranteed to be preserved for the next entry into the block. STATIC data is not reinitialized whenever a program is reentered (executed again). Values are preserved in this way even though a STATIC <identifier> has no <initialization>. Preservation of values is not guaranteed for AUTOMATIC <identifier>s. If neither keyword appears, then STATIC is assumed.
   - DENSE/ALIGNED - The appearance of one of these keywords is mutually exclusive of the other. Although legal in other contexts, the keywords are only effective when appearing as <attributes> in a <structure template>. DENSE and ALIGNED refer to the storage packing density to be employed when a <structure var name> is declared using the template. If neither keyword appears, then ALIGNED is assumed.
   - ACCESS - This attribute causes implementation dependent managerial restrictions to be placed upon the usage of the <identifier> as a variable in assignment contexts. The manner of enforcement of the restrictions is implementation dependent.
   - LOCK - This attribute causes use of the <identifier> to be restricted to the interior of UPDATE blocks, and to assign argument lists. <number> indicates the “lock group” of the <identifier> and lies between 1 and an implementation-dependent maximum. “*” indicates the set of all lock groups. Specifying LOCK(*) for a formal parameter overrides the LOCK attribute (if any) of the corresponding argument in the invocation. The purpose of the attribute is described in Section 8.10.
   - LATCHED - See Section 4.7.
   - <initialization> - This attribute describes the manner in which the values of an <identifier> are to be initialized. It is described in Section 4.8.
   - REMOTE - This attribute identifies data which is to be located in areas separate from normal data. Its implementation is machine dependent. Its purpose is to provide information to the compiler so that proper addressing to the data can be generated. Generally, this addressing requires longer and slower access methods. REMOTE data (not NAME variables) cannot be AUTOMATIC within a REENTRANT procedure.

- RIGID - Although legal in other contexts, the keyword is only effective when appearing as an <attribute> in a <structure template> or in a COMPOOL. It causes data to be organized in the order it is defined within the <structure template>.

RESTRICTIONS FOR SIMPLE VARIABLES AND MAJOR STRUCTURES:

1. The asterisk form of array specification can only be applied to an <identifier> if it is a formal parameter of a procedure or function. The actual length of the array is supplied by the corresponding argument of an invocation of the procedure or function.
2. An array specification is illegal if the <identifier> is defined by the <type spec> to be a major structure.
3. The ACCESS attribute may only be applied to <identifier> names declared in a <compool block> or <compool template>. The LOCK attribute may only be applied to <identifier> names declared in a <compool block>, <compool template>, <program block>, or to the assign parameters of procedure blocks.
4. The LATCHED attribute only applies to EVENT variables (see Section 4.7).
5. The REMOTE and AUTOMATIC attributes are illegal for any <identifier> of EVENT type. AUTOMATIC is also illegal if <identifier> is a parameter of a PROCEDURE or FUNCTION block.
6. The attributes ALIGNED and RIGID are illegal for major structures.
7. The <initialization> attribute may not be applied to formal parameters of procedures and functions.

RESTRICTIONS FOR STRUCTURE TERMINALS:

1. The asterisk form of array specification is not allowed.
2. The <identifier> may not be defined to be an unqualified structure by the <type spec>. Otherwise, the type specification is the same as for simple variables.
3. The appearance of any minor attributes except DENSE, ALIGNED, and RIGID is illegal. Appearances of DENSE and ALIGNED override such appearances on the minor structure levels or on the <structure template> name itself.
4. An array specification is illegal if the <identifier> is defined by the <type spec> to be a major structure.

RESTRICTIONS FOR MINOR STRUCTURES:

1. The <type spec> for a minor structure name must be empty (see Section 4.7).
2. No array specification is allowed.
3. No attributes except DENSE, ALIGNED, and RIGID are allowed. Appearances of DENSE and ALIGNED at any level of the structure override such appearances at higher levels or on the <structure template> name itself. The appearance of RIGID causes structure terminals within the minor structure to be organized in the order in which they are declared. However, RIGID at the minor structure level will not affect the order of data within an included template specified by a structure template <typespec>.

```
EXAMPLE:
  STRUCTURE Y:
    1 A SCALAR,
    1 B VECTOR(4),
    1 D MATRIX(4,4);
  STRUCTURE Z RIGID:
    1 F BIT(13),
    1 G Y-STRUCTURE,
    1 H CHARACTER(10);
```

The order within Z will be: F, G, H, but the order within G will not necessarily be as declared by Y.

Figure 4-11 Rigid Structure Example

4.6 Label Declarative Attributes.

A label declarative attribute defines an <identifier> to be a <label> of some specific type.

SYNTAX:

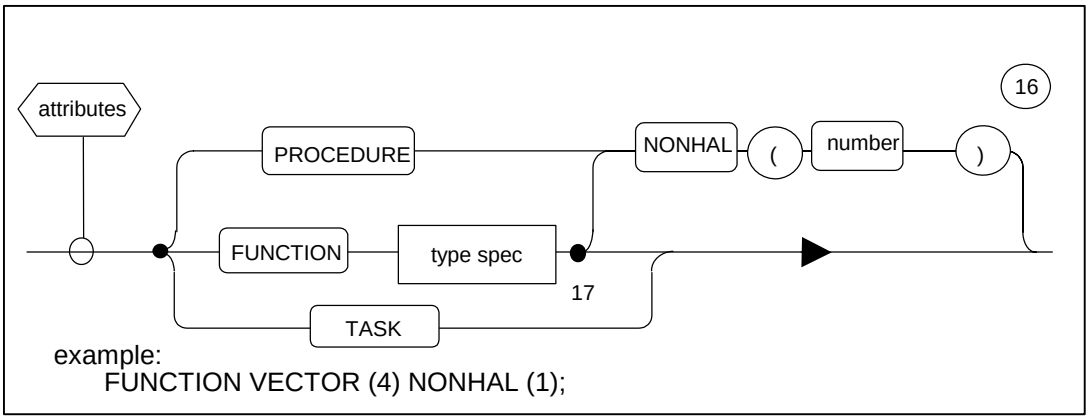


Figure 4-12 Label declarative attributes - #16

SEMANTIC RULES:

1. The form FUNCTION <type spec> is used to define the name and type of a <function block>. Such a definition is only required if the function is referenced in the source before the occurrence of its block definition.

   Functions requiring definition this way are subject to the following restrictions:

   - they must have at least one formal parameter;
   - none of their formal parameters may be arrayed.

   The type specification of the function declared is given by <type spec> (see Section 4.7). A function may be of any type except EVENT.

2. The NONHAL (<number>) indicates that an external routine written in some other language is being declared. The NONHAL (<number>) may be a factored attribute applied to a list of label declarations. The <number> is an implementation dependent indication of the type of NONHAL linkage.
3. The form TASK is used to define the name of a <task block>. It may be required if a <task block> is referenced before the occurrence of its definition.

4.7 Type Specification.

The type specification or <type spec> provides a means of defining the type (and precision where applicable) of data names and parts of structure templates.

SYNTAX:

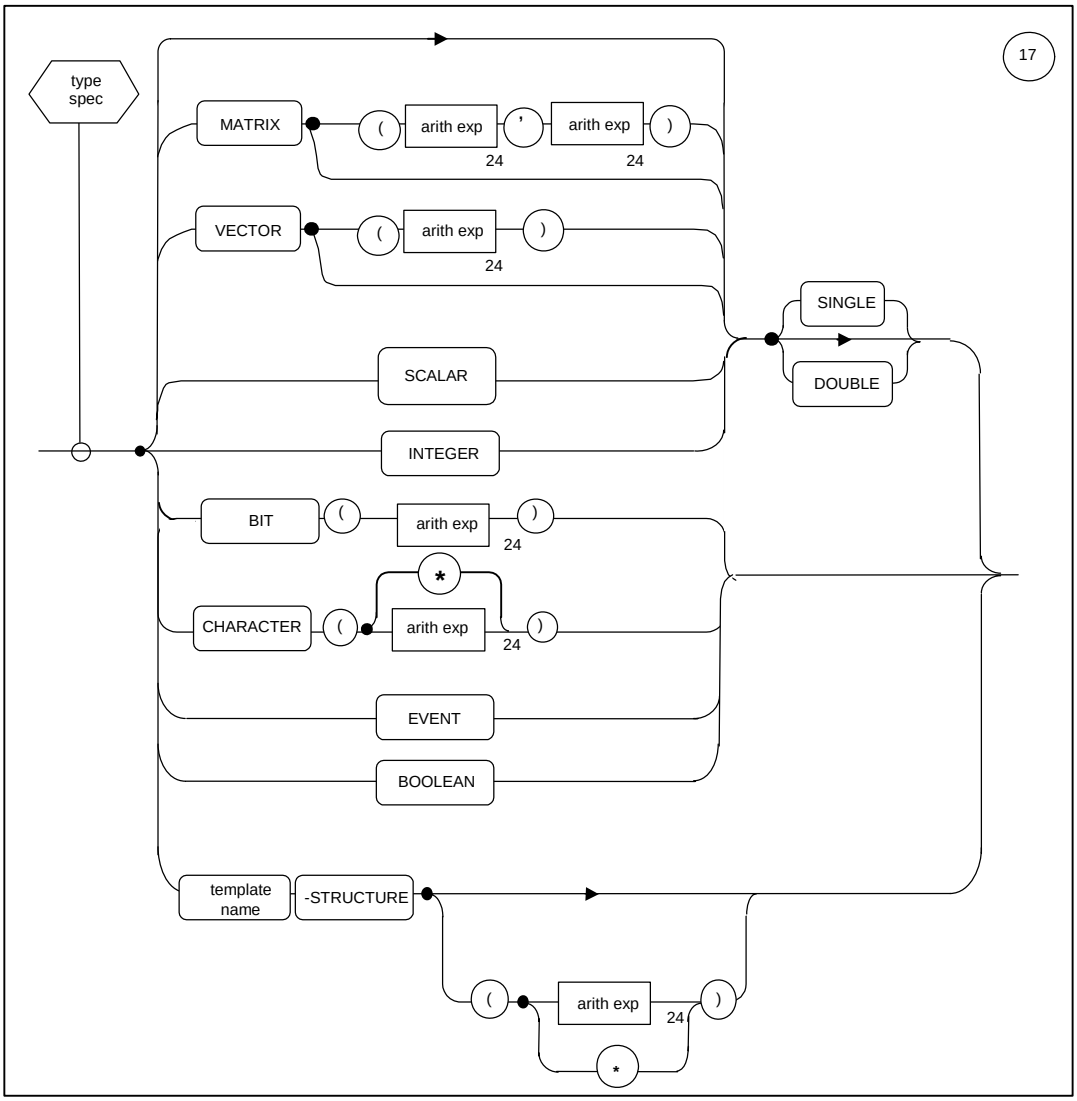


Figure 4-13 type specification - #17

GENERAL SEMANTIC RULES:

1. If <type spec> is empty (i.e., there is no specification present) then the interpretation is as follows:
   - if the <type spec> is that of a simple variable, function, or structure terminal, then the implied type is SCALAR with SINGLE precision.
   - otherwise the <type spec> is that of a minor structure of a structure template.
2. If the <type spec> is empty except for the keyword SINGLE or DOUBLE, the implied type is SCALAR with the indicated precision.
3. The precision keywords only apply to VECTOR, MATRIX, SCALAR, and INTEGER <type spec>s. In the last case SINGLE implies a halfword integer, and DOUBLE a fullword integer. In the absence of a precision keyword, SINGLE is presumed.
4. Any <arith exp> in a <type spec> is an unarrayed integer or scalar expression computable at compile time (see Appendix F). Its value is rounded to the nearest integer (specifying a scalar expression is exactly the same as specifying its integer equivalent.).

RULES FOR INTEGER AND SCALAR TYPES:

1. Integer and scalar types are indicated by the keywords INTEGER and SCALAR, respectively. Note that scalar type can be indicated implicitly as described in General Semantic Rules 1 and 2.

RULES FOR VECTOR AND MATRIX TYPES:

1. The keyword MATRIX is used to indicate matrices containing scalar components. If present, the two <arith exp>s in parentheses give the row and column dimensions of the matrix, respectively. In the absence of such a size specification, MATRIX (3,3) is implied.
2. The keyword VECTOR is used to indicate vectors containing scalar components. If present, the parenthesized <arith exp> indicates the length of the vector. In the absence of a length specification, VECTOR(3) is implied.
3. The row and column dimensions of a matrix, and the length of a vector may range between 2 and an implementation dependent maximum.

RULES FOR CHARACTER TYPES:

1. Character type is indicated by the keyword CHARACTER. A character variable is of varying length; the parenthesized <arith exp> following the keyword CHARACTER denotes the maximum length that the character variable may take on. A length must be specified.
2. The working length of a character data type may range from zero (the “null” string) to the defined maximum length.
3. The defined maximum length has an upper limit which is implementation dependent.
4. The asterisk form of character maximum length specification must be applied to an <identifier> if it is a formal parameter of a procedure or function. The actual length information of the character string is supplied by the corresponding argument in the invocation of the procedure or function.

RULES FOR BIT, BOOLEAN, AND EVENT TYPES:

1. The keyword BIT indicates type. The following parenthesized <arith exp> gives the length in bits. Its value may range between 1 and an implementation dependent upper limit.
2. The keyword BOOLEAN indicates a bit type of 1-bit length.
3. The keyword EVENT indicates an event type, similar to BOOLEAN, but which differs in that it has real time programming implications (see Section 8). An <identifier> of event type is the only type to which the attribute LATCHED is applicable. The implications of the LATCHED attribute are discussed in Section 8.8. An <identifier> of event type may not be used as an input formal parameter, nor may it be a structure terminal.

RULES FOR STRUCTURE TYPE:

1. The condition for the <type spec> indicating a minor structure is described in General Semantic Rule 1.
2. The phrase <template name>-STRUCTURE defines an <identifier> to be a major structure whose tree organization is described by a previously defined template called <template name>.
3. The parenthesized expression or asterisk optionally following the keyword STRUCTURE specifies the structure to have multiple copies. The value specifies the number of copies, which may range from 2 to an implementation dependent maximum.
4. The copy specification may only be an asterisk if the structure is a formal parameter of a procedure or function. The actual number of copies is supplied by the corresponding argument of an invocation of the procedure or function and must be greater than 1.
5. If the <identifier> named is the same as the <template name> of the template of the structure, then the structure is said to be unqualified; otherwise, the structure is said to be qualified. Templates used for non-qualified declarations may not contain nested structure references. Section 5.2 contains material on some further implications of structure qualification.[6]
6. If the <type spec> of a function is STRUCTURE, then no specification of multiple copies is allowed.
7. If the <type spec> of a structure terminal is STRUCTURE, then no specification of multiple copies is allowed.

4.8 Initialization.

The <initialization> attribute specifies the initial values to be applied to an <identifier>. The circumstances under which the attribute is legal have been described in Section 4.5.

6. Declarations of non-qualified STRUCTUREs must occur in the same name scope as the template definition.

SYNTAX:

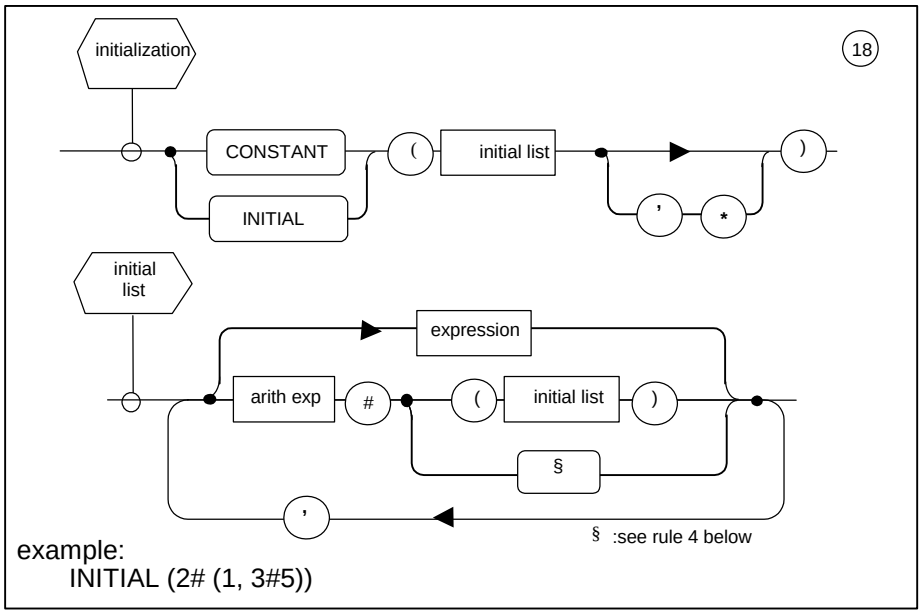


Figure 4-14 initialization specification - #18

GENERAL SEMANTIC RULES

1. The <initialization> starts with the keyword INITIAL or CONSTANT. If it starts with CONSTANT, the value of the <identifier> initialized may never be changed. It is illegal for <identifier>s with CONSTANT <initialization> to appear in an assignment context.
2. The simplest form of an <initial list> is a sequence of one or more <expression>s computable at compile time (see Appendix F).
3. A simple <initial list> of the form given in Rule 2 may appear enclosed in parentheses, and preceded by <arith exp>#, where <arith exp> is any unarrayed integer or scalar expression computable at compile time. The value, rounded to the nearest integer, is a repetition factor for the initial values contained within the parentheses. This repeated <initial list> may itself become a component of an <initial list>, and so on to some arbitrary nesting depth.
4. In addition to preceding a parenthesized <initial list>, <arith exp># may also precede certain unparenthesized items denoted collectively in the syntax diagram by §. These items are:
   - a single literal;
   - a single unsubscripted variable name;
   - blank (i.e., the component(s) of the <identifier> should not be initialized).
5. The presence of an asterisk at the end of an <initial list> implies the partial initialization of an <identifier>.
6. The order of initialization is the “natural sequence” specified in Section 5.5.

RULES FOR INTEGER AND SCALAR TYPES:

1. If the <identifier> has no array specification, the <initial list> must contain exactly one value.
2. If the <identifier> has an array specification, then one of the following must hold:
   - the number of values in the <initial list> is exactly one, in which case all elements of the array are initialized to that value;
   - the number of values in the <initial list> is exactly equal to the number of array elements to be initialized;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of array elements to be initialized, and partial initialization is indicated.
3. <expression> must be an unarrayed INTEGER or SCALAR, or expression computable at compile time (see Appendix F). Type conversion between INTEGER and SCALAR is allowed where necessary.
4. Non-aggregate (unarrayed and not in a structure) INTEGER and SCALAR single precision CONSTANTs initialized with a double precision value remain double precision.

RULES FOR VECTOR AND MATRIX TYPES:

1. If the <identifier> has no array specification, then one of the following must hold:
   - the number of values in the <initial list> is exactly one, in which case all components of the VECTOR or MATRIX are initialized to the value;
   - the number of values in the <initial list> is exactly equal to the number of components to be initialized;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of components to be initialized, and partial initialization is indicated.
2. If the <identifier> has an array specification, then one of the following must hold:
   - the number of values in the <initial list> is exactly one, in which case all the components of all the array elements of the VECTOR or MATRIX are initialized to that value;
   - the number of values in the <initial list> is exactly equal to the number of components of the VECTOR or MATRIX, in which case every array element takes on the same set of values;
   - the number of values in the <initial list> is equal to the total number of components in all array elements;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the total number of components in all array elements, and partial initialization is indicated.
3. <expression> must be an unarrayed integer or scalar expression computable at compile time (see Appendix F). The conversion between integer and scalar is allowed where necessary.

RULES FOR BIT, BOOLEAN, EVENT, AND CHARACTER TYPES:

1. If the <identifier> has no array specification, the <initial list> must contain exactly one value.
2. If the <identifier> has an array specification, then one of the following must hold:
   - the number of values in the <initial list> is exactly one, in which case all elements of the array are initialized to that value;
   - the number of values in the <initial list> is exactly equal to the number of array elements to be initialized;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of array elements to be initialized, and partial initialization is indicated.
3. If an <identifier> of Bit, Boolean, or Event type is being initialized, <expression> must be an unarrayed <bit exp> computable at compile time (see Appendix F). If an Event <identifier> has the LATCHED attribute, then it may be initialized to the value TRUE or FALSE (or their equivalent). If it does not have the LATCHED attribute, then it cannot be initialized (see Section 8.8). In the absence of <initialization> all Events are implicitly initialized to FALSE and all Bit variables are set to zero. If the memory location containing the Bit type is larger than the declared size of the Bit <identifier>, then the bits that do not correspond to an <identifier> will also be initialized to zero.
4. If an <identifier> of CHARACTER type is being initialized, <expression> must be an unarrayed <char exp> computable at compile time (see Appendix F).

RULES FOR STRUCTURE TYPES:

1. Only a major structure <identifier> may be initialized.
2. If the <identifier> has only one copy, then one of the following must hold:
   - the number of values in the <initial list> is equal to the total number of data elements in the whole structure;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of data elements in the whole structure, and partial initialization is indicated.
3. If the <identifier> has multiple copies, then one of the following must hold:
   - the total number of values in the <initial list> is exactly equal to the total number of data elements in one copy of the structure, in which case each copy is identically initialized;
   - the number of values in the <initial list> is equal to the total number of data elements in all copies of the structure;
   - the <initial list> ends with an asterisk, in which case the number of values must be less than the total number of data elements in all copies of the structure, and partial initialization is indicated.

The type of each <expression> must be legal for the type of corresponding structure terminal initialized (see the Semantic Rules for initialization of simple variables of each type).

5.0 DATA REFERENCING CONSIDERATIONS

Central to the HAL/S language is the ability to access and change the values of variables. Section 4 dealt comprehensively with the way in which data names are defined. This section addresses itself to the various ways these names can be compounded and modified when they are referenced.

5.1 Referencing Simple Variables.

In Section 4.5 the term “simple variable” was introduced to describe a data name which was not a structure, or part of one. When a simple variable is defined in a <declare group>, it is syntactically denoted by the <identifier> primitive. Thereafter, since its attributes are known, it is denoted syntactically by the <§ var name> primitive, where § stands for any of the types arithmetic, bit, character, or event.

5.2 Referencing Structures

When an <identifier> is declared to be a structure, its tree organization is that of the template whose <template name> appears in the structure declaration (see Section 4.7). References to the structure as a whole (the “major structure”), are obviously made by using the declared <identifier>, which syntactically becomes a <structure var name>. The way in which parts of the structure (its minor structures and terminals) are referenced depends on whether the structure is “qualified” or “unqualified” (see Section 4.7).

- If a structure is “unqualified”, then any part of it, either minor structure or structure terminal, may be referenced by using the name of the part as it appears in the <structure template>. If a minor structure is referenced, the name becomes syntactically a <structure var name>. If a structure terminal is referenced, then syntactically the name becomes a <§var name>, where § stands for any of the types arithmetic, bit, character, or event, as specified in its <attributes> in the template.
- If the structure is “qualified”, then any part of it, either minor structure or structure terminal, is referenced as follows. First the major structure name is taken. Then starting at the template name, the branches of the template are traversed down to the minor structure or structure terminal to be referenced. On passing through every intervening minor structure, the name is compounded by right catenating a period followed by the name of the minor structure passed through. The process ends with the catenation of the name of the minor structure or structure terminal to be referenced. If a minor structure is being referenced, the resulting “qualified” name becomes syntactically a <structure var name>. If a structure terminal is referenced, then syntactically it becomes a <§var name>, where § stands for any of the types arithmetic, bit, character, or event, as specified in its <attributes> in the template.

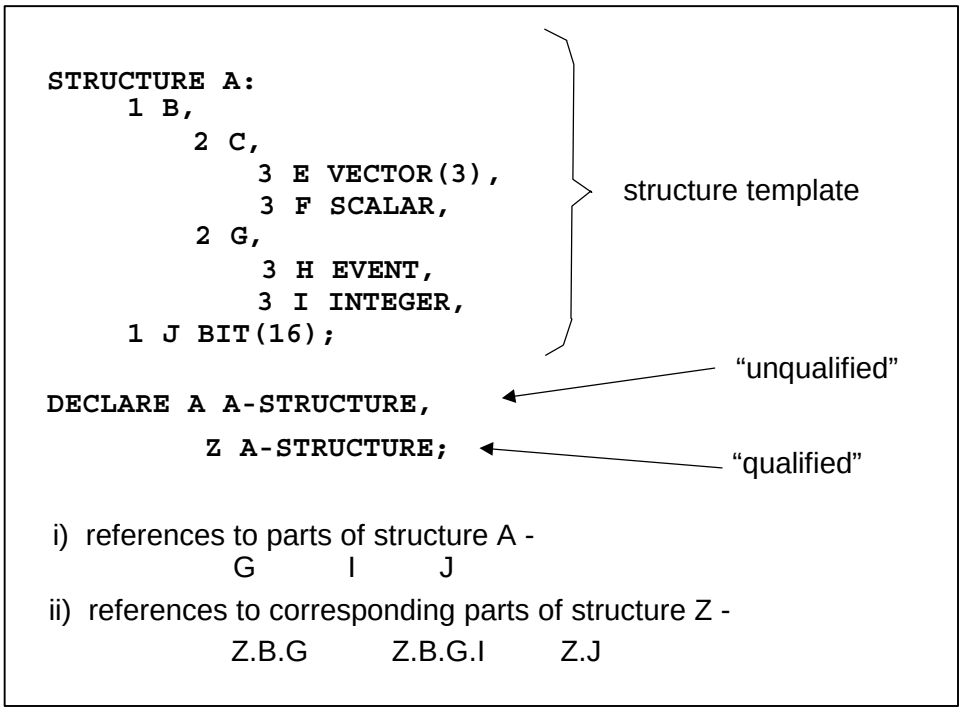


Figure 5-1 Referencing Structures

5.3 Subscripting

For the remainder of the section, a data name with known <attributes> is denoted syntactically by <§var name>, where § stands for any of the types arithmetic, bit, character, event, or structure. It is convenient to introduce the syntactical term <§var> to denote any subscripted or unsubscripted <§var name>.

SYNTAX:

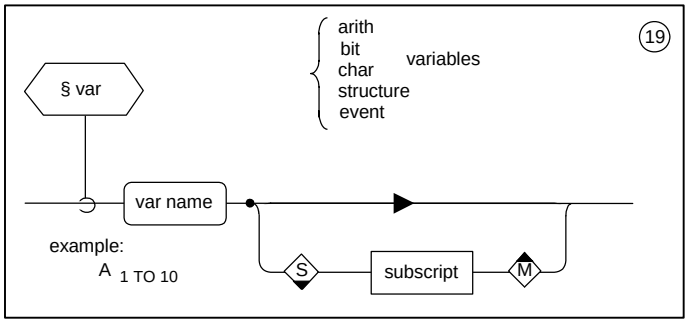


Figure 5-2 Subscripting - #19

It is also useful to introduce the syntactical term <variable> as a collective definition meaning any type of <§var>.

SYNTAX:

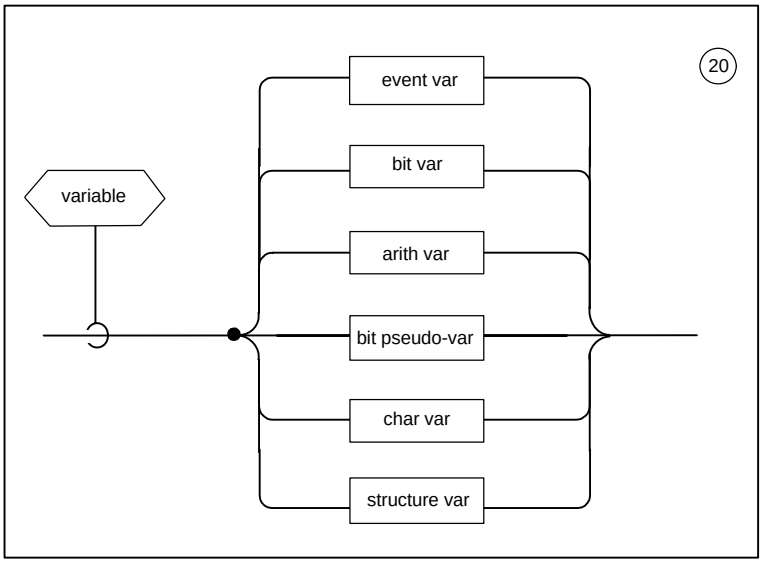


Figure 5-3 variable - #20

SEMANTIC RULES

1. <bit pseudo-var> is a reference to the SUBBIT pseudo-variable. An explanation of its inclusion as a <variable> is given in Section 6.5.4.

5.3.1 Classes of Subscripting

In HAL/S, there are three classes of subscripting which may be potentially applied to <§var name>s: structure, array, and component subscripting.

- Structure subscripting can be applied to arithmetic, bit, character, and event variables which are terminals of a structure which has multiple copies. It can also be applied to the major and minor structure variable names of such a structure. Structure subscripting is denoted syntactically by <structure sub>.
- Array subscripting can be applied to any arithmetic, bit, character, and event variables which are given an array specification in their declaration. This includes both simple variables and structure terminals. Array subscripting is denoted syntactically by <array sub>.
- Component subscripting can be applied to simple variables and structure terminals which have one or more component dimensions (i.e., which are made up of distinct components). The applicable types are vector, matrix, bit, and character. Component subscripting is denoted syntactically by <component sub>.

The three classes of subscript are combined according to a well-defined set of rules.

SYNTAX:

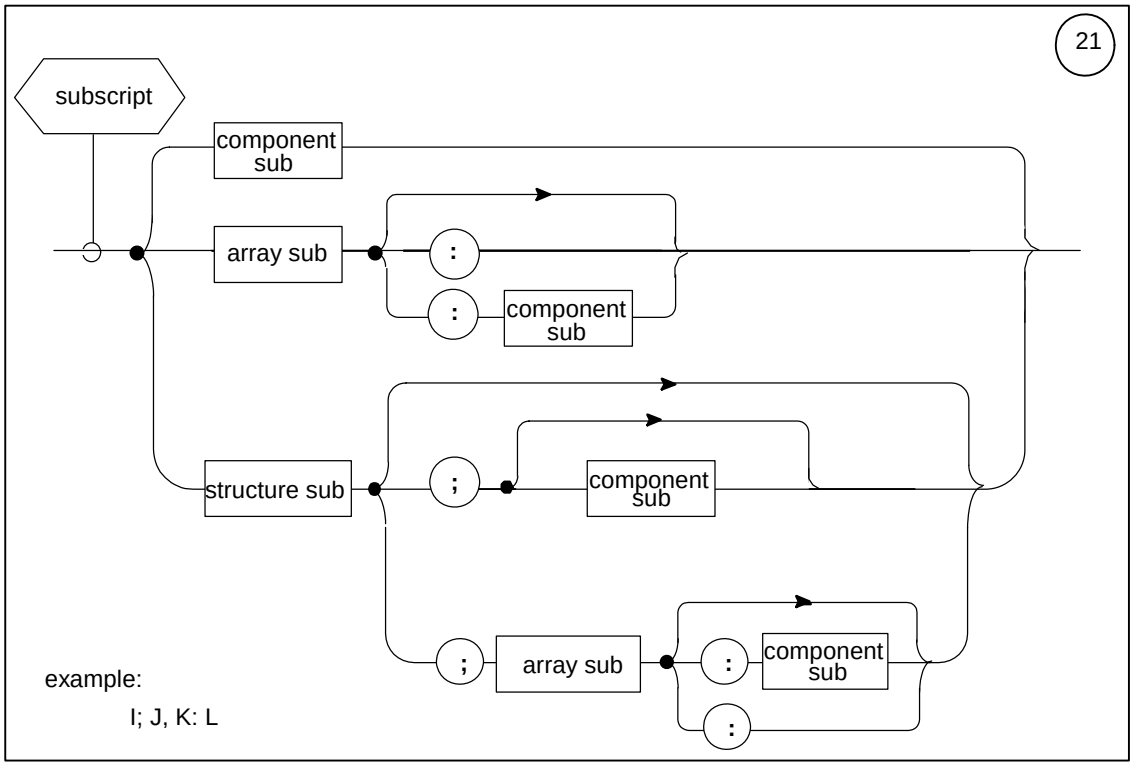


Figure 5-4 subscript construct - #21

SEMANTIC RULES:

1. The syntax diagram shows 10 different ways of combining the three classes of subscripting. The following table shows when each of these combinations is legal for simple variables and structure terminals. In the table, the following abbreviations are used:

   <component sub> → C
   <array sub> → A
   <structure sub> → S

Interpretation of <§ var name>				
data type	unarrayed simple variable	arrayed simple variable	unarrayed structure terminal ①	arrayed structure terminal ①
integer scalar event	none	A A:	S S;	S; S; A S; A:
vector matrix bit character	C	A: A: C	S; S; C	S; S; A S; A: C

① It is assumed that the structure has multiple copies. If not, corresponding columns for simple variables apply.

2. In the case of a <structure var name> relating to a major structure with multiple copies, or to a minor structure of such a major structure, the following forms are legal:

S
S;

No subscript is possible if the major structure has no multiple copies.

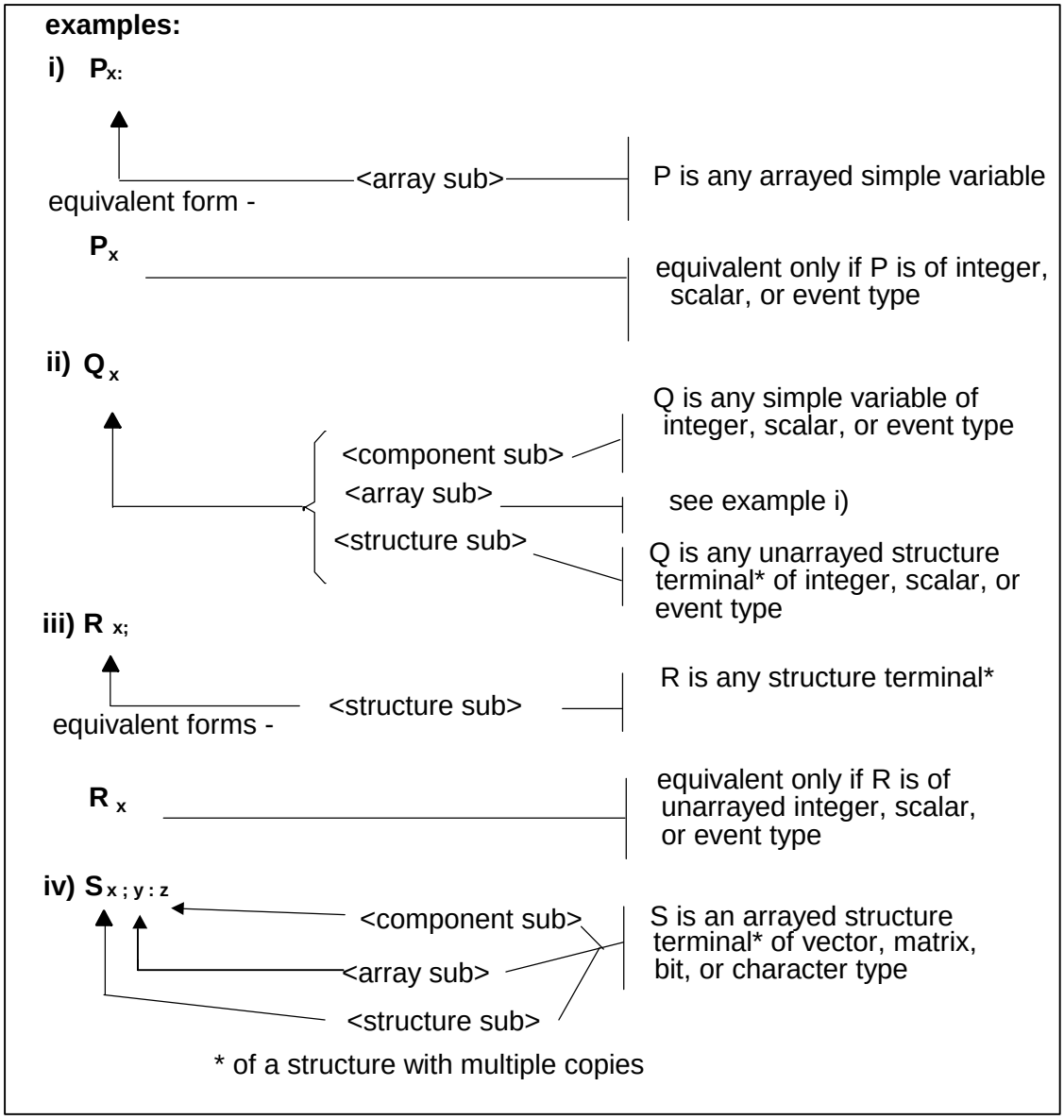


Figure 5-5 Subscript examples

5.3.2 The General Form of Subscripting.

The three classes of subscripting, <structure sub>, <array sub>, and <component sub>, have an identical syntactical form; however, the semantic rules for each differ.

SYNTAX:

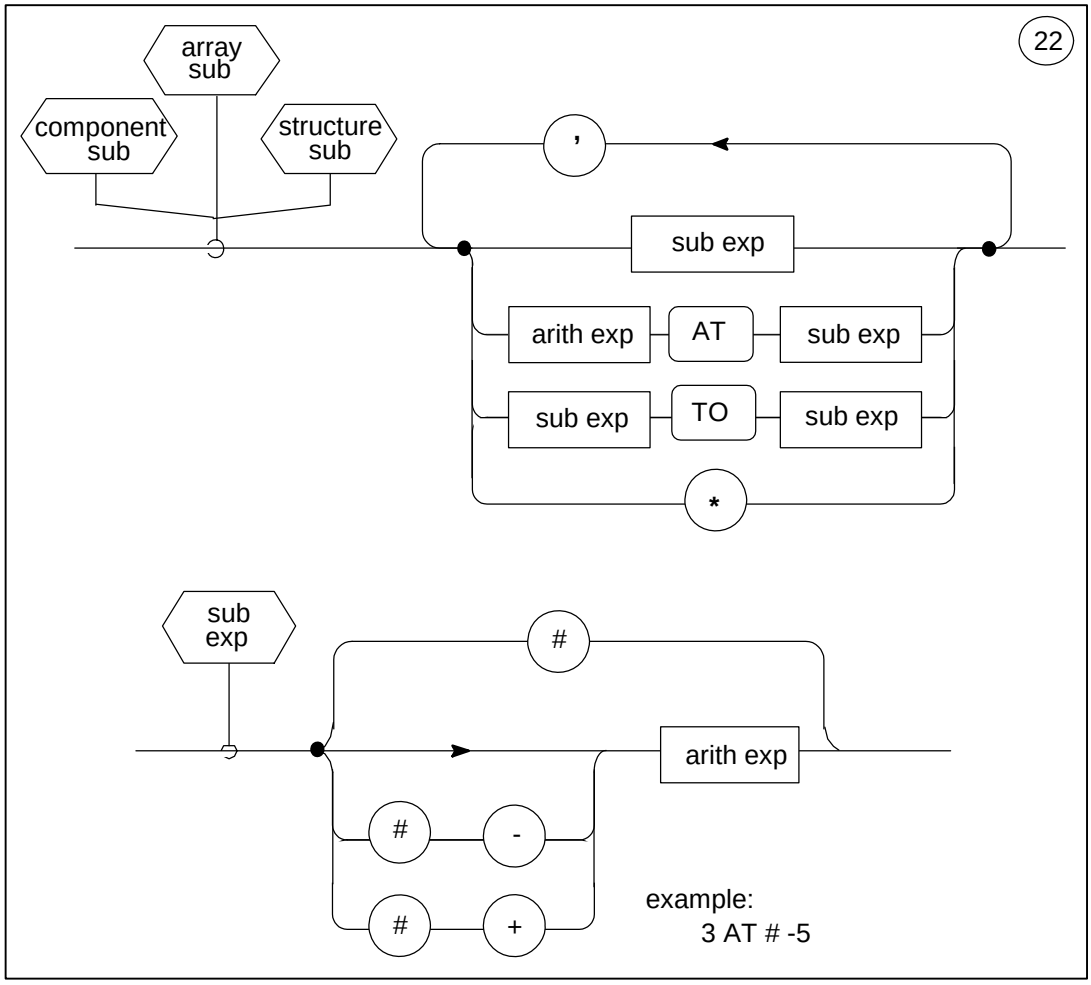


Figure 5-6 componoent, array, and structure subscripts - #22

GENERAL SEMANTIC RULES:

1. A <structure sub>, <array sub>, or <component sub> consists of a series of "subscript expressions" separated by commas. Each subscript expression corresponds to a structure, array, or component dimension of the <§ var name> subscripted.
2. There are four forms of subscript expression:
   - the simple index;
   - the AT-partition ;
   - the TO-partition;
   - the asterisk.

3. The simple index form is denoted in the diagram by a single <sub exp>. Its value specifies the index of a single component, array element, or structure copy to be selected from a dimension.
4. The AT-partition is denoted by the form <arith exp> AT <sub exp>. The value of <arith exp> is the width of the partition, and that of <sub exp> the starting index.
5. The TO-partition is denoted by the form <sub exp> TO <sub exp>. The two <sub exp> values are the first and last indices, respectively, of the partition.
6. The asterisk form, denoted in the diagram by *, specifies the selection of all components, elements, or copies from a dimension.
7. <sub exp> may take any of the forms shown. The value of # is taken to be the maximum index-value in the relevant dimension (for character variables, this is the current length).
8. Any <arith exp> in a subscript expression is an unarrayed integer or scalar expression. Values are rounded to the nearest integer.

5.3.3 Structure Subscripting.

Major structures with multiple copies, or the minor structures or structure terminals of such structures may possess a <structure sub>. Since there is only one dimension of multiple copies, the <structure sub> may only possess one subscript expression. The effect of such subscripting is to eliminate multiple copies, or at least to reduce their number.

RESTRICTIONS:

1. Errors result if any index value implied by a subscript expression lies outside the range 1 through *N*, where *N* is the number of copies specified for the major structure.
2. If the subscript expression is a TO- or AT-partition, the width of the partition must be computable at compile time. This is guaranteed by enforcing the following restrictions:
   - in the form <arith exp> AT <sub exp>, the value of the <arith exp> must by computable at compile time (see Appendix F).
   - in the form <sub exp> TO <sub exp>, the values of both <sub exp>s must be computable at compile time.

examples:

```
STRUCTURE A;
 1 B SCALAR,
 1 C INTEGER,
 1 D VECTOR(6);
 •
 •
 •
DECLARE A A-STRUCTURE(20);
```

A_{20}	20^{th} copy of A
$A_{2\ AT\ 10;}$	10^{th} and 11^{th} copies of A (semicolon optional)
C_{1}	C from 1^{st} copy of A
$D_{4\ TO\ 6;}$	D from 4^{th} through 6^{th} copies of A (semicolon enforced)
Note: $D_{*;4\ TO\ 6}$	components 4 through 6 of D from all copies of A

Figure 5-7 Structure Subscripting Examples

5.3.4 Array Subscripting.

Any simple variable or structure terminal with an array specification (see Section 4.5) may possess an <array sub>. The number of subscript expressions in the <array sub> must equal the number of dimensions given in the array specification. The leftmost subscript expression corresponds to the leftmost dimension of the array specification, the next expression to the next dimension, and so on.

RESTRICTIONS:

1. Errors result if any index value implied by a subscript expression lies outside the range 1 through *N*, where *N* is the size of the corresponding dimension in the array specification.
2. If the subscript expression is a TO- or AT-partition, the width of the partition must be computable at compile time. This is guaranteed by enforcing the following restrictions:
   - in the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time;
   - in the form <sub exp> TO <sub exp>, the value of both <sub exp>s must be computable at compile time.

examples

```
     STRUCTURE P:
       1 Q ARRAY(5) SCALAR,
       1 R SCALAR;
      •
      •
      •
     DECLARE P P-STRUCTURE(10);

     DECLARE S ARRAY (5) SCALAR,
              T ARRAY (5) VECTOR(6);
```

	$Q_{*;5}$	5th array element of Q in all copies of P
	$Q_{1;2\ TO\ 3:}$	2nd and 3rd array elements of Q in 1st copy of P (colon optional)
	$S_{4\ TO\ 5:}$	4th through 5th array elements of S (colon optional)
	$T_{2\ AT\ 2:}$	2nd and 3rd array elements of T (colon enforced)
Note:	$T_{*:2\ AT\ 2}$	components 2 and 3 in all array elements of T

Figure 5-8 Structure and Array Subscripting Examples

5.3.5 Component Subscripting.

Simple variables and structure terminals of vector, matrix, bit, and character type may possess component subscripting because they are made up of multiple distinct components.

- Those of bit, character, and vector types must possess a <component sub> consisting of one subscript expression only;
- Those of matrix type must possess a <component sub> consisting of two subscript expressions. In left to right order these represent row and column subscripting respectively.

RESTRICTIONS:

1. Errors result if any index value implied by a subscript expression lies outside the range 1 through *N*, where *N* is the size of the corresponding dimension in the type specification.
2. For bit, vector, and matrix types, if the subscript is TO- or AT-partition, the width of the partition must be computable at compile time. This is guaranteed by enforcing the following restrictions:
   - in the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time;
   - in the form <sub exp> TO <sub exp>, the value of both <sub exp>s must be computable at compile time.

3. The subscript expressions of a character type need not be computable at compile time.

SPECIAL RULES FOR VECTOR AND MATRIX TYPES:

The <component sub> of a variable of vector or matrix type can sometimes have the effect of changing its type. The following rules apply:

1. If a VECTOR type is subscripted with a simple index <component sub>, then since one component is being selected, the resulting <arith var> is of SCALAR type.
2. If only one of the two subscript expressions in a <component sub> of a MATRIX type is a simple index, then one row or column is being selected, and the result is therefore an <arith var> of VECTOR type. If both subscript expressions are of simple index form, then one component of the MATRIX is being selected, and the result is an <arith var> of SCALAR type.

examples:

```
DECLARE M MATRIX(3,3),
        C ARRAY(2) CHARACTER(8);
 .
 .
 .
```

$C_{1:2 \text{ TO } 7}$	characters 2 through 7 of 1st array element of C
$M_{*,1}$	column 1 of matrix M (vector)
$M_{3,3}$	3rd component of 3rd row of M (scalar)

Figure 5-9 Matrix and Array Subscripting Examples

5.4 The Property of Arrayness.

A <§var name> which is a simple variable is said to be “arrayed”, or to possess “arrayness”, if any array specification appears in its declaration. The number of dimensions of arrayness is the number of dimensions given in the array specification.

A <§var name> which is a structure terminal is said to be arrayed or to possess arrayness if either or both of the following hold:

- an array specification appears in its declaration in a structure template;
- the structure of which <¤var name> is a terminal has multiple copies.

The number of dimensions of arrayness is the sum of the dimensions originating from each source.

Appending structure or array subscripting to a <§var name> may reduce the number and size of array dimensions of the resulting <§var>.

The arrayness of HAL/S expressions originates ultimately from the <§var>s contained in them. It is a general rule that all arrayed <§var>s in an expression must possess identical arrayness (i.e. the number of dimensions of arrayness, and their corresponding sizes must be the same). Although the forms of subscript distinguish between array dimensions, and structure copy dimensions, no distinction between them is made as far

as the matching of arrayness is concerned.

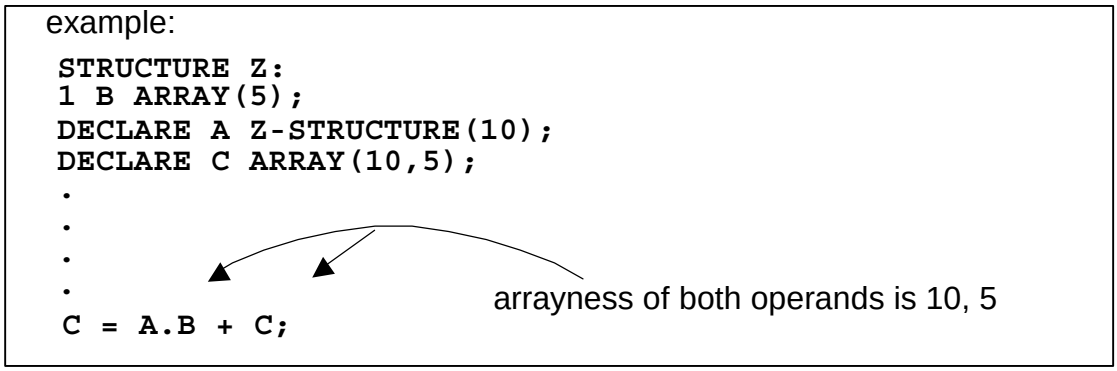


Figure 5-10 Property of Arayness Exapmle

5.4.1 Arrayness of Subscript Expressions

Any <arith exp> within a subscript may be arrayed (possess "arrayness"). Appending such subscripts to a <§var name> may produce an arrayed operand of the same arrayness as the <arith exp>. The following rules are applicable to such subscript forms.

SEMANTIC RULES:

1. Any <arith exp> appearing in Syntax Diagram 22 depicting the syntax of <structure sub>, <array sub>, and <component sub> may potentially possess arrayness.
2. If the <§var name> possessing the subscript containing the arrayed <arith exp> is imbedded in an arrayed HAL/S expression, then the arrayness of the <arith exp> must match the arrayness of the expression, even if the <var name> itself does not possess arrayness (e.g. is a vector).
3. The evaluation of an arrayed expression can be viewed as a parallel evaluation of the expression element by element. If the expression contains an arrayed <arith exp> in a subscript, then during the parallel evaluation the appropriate array element of <arith exp> is selected for each evaluation.

Example:

Given the declarations:

```
DECLARE A ARRAY(3) INTEGER;
DECLARE B ARRAY(3,2) INTEGER;
DECLARE V VECTOR(5);
```

the following operands become:

V_A - a 3 - array
V_B - a 3x2 - array } of corresponding vector components

Example

```
DECLARE I ARRAY(3) INTEGER,
    M MATRIX(2,2),
    MA ARRAY(3)MATRIX(2,2),
    MB ARRAY(2)MATRIX(2,2);
```

Let $M \equiv \begin{bmatrix} 1.75 & 0.25 \\ 0.75 & 1.25 \end{bmatrix}$ and $I \equiv \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$

then $M_{I,*} \equiv \begin{pmatrix} M_{2,*} \\ M_{1,*} \\ M_{1,*} \end{pmatrix} \equiv \begin{pmatrix} [0.75 \;\; 1.25] \\ [1.75 \;\; 0.25] \\ [1.75 \;\; 0.25] \end{pmatrix}$

- a linear 3-array of 2-vectors: subscripting has reduced M from a matrix to a row-vector, but since I is arrayed, the entire operand has an effective arrayness even though M itself has not.

Let $MA \equiv \begin{pmatrix} \begin{bmatrix} 1.0 & 0.0 \\ 3.0 & 2.0 \end{bmatrix} \\ \begin{bmatrix} 4.0 & 7.0 \\ 6.0 & 5.0 \end{bmatrix} \\ \begin{bmatrix} 8.0 & 3.0 \\ 4.0 & 9.0 \end{bmatrix} \end{pmatrix}$

① $I_1 = 2$

② $I_2 = 1$

③ $I_3 = 1$

Then $MA_{*:I,*} \equiv \begin{bmatrix} (M_{1:2,*}) \\ (M_{2:1,*}) \\ (M_{3:1,*}) \end{bmatrix} \equiv \begin{pmatrix} [3.0 \;\; 2.0] \\ [4.0 \;\; 7.0] \\ [8.0 \;\; 3.0] \end{pmatrix}$

is also a linear 3-array of 2-vectors: now however MA and I both have arrayness (which correctly match). Three parallel subscript evaluations are effectively performed using corresponding array elements of MA and I each time.

Note $MB_{*:I,*}$ is illegal since the arrayness of MB does not match the arrayness of I.

However $MB_{*:I\ 1\ TO\ 2,*}$ is legal since array subscripting has been used on I to force arrayness matching.

If $MB \equiv \begin{pmatrix} \begin{bmatrix} 0.5 & 0.5 \\ 0.1 & 0.3 \end{bmatrix} \\ \begin{bmatrix} 0.2 & 0.7 \\ 0.4 & 0.8 \end{bmatrix} \end{pmatrix}$

① $I_1 = 2$

② $I_2 = 1$

then $MB_{*:I\ 1\ TO\ 2,*} \equiv \begin{pmatrix} MB_{1:2,*} \\ MB_{2:1,*} \end{pmatrix} \equiv \begin{pmatrix} \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.7 \end{bmatrix} \end{pmatrix}$

5.5 The Natural Sequence of Data Elements

There are several kinds of operations in the HAL/S language which require operands with multiple components, array elements, and structure copies to be unraveled into a linear string of data elements. The reverse process of "reraveling" a linear string of data elements into components, array elements, and structure copies also occurs. The two major occurrences of these processes are in I/O (see Section 10) and in conversion functions (see Section 6.5).

The standard order in which this unraveling and reraveling takes place is called the "natural sequence". By applying the following rules in the order they are stated, the natural sequence of unraveling is obtained. By applying the rules in reverse order, and by replacing "unraveled" by "reraveled", the natural sequence for reraveling is obtained.

RULES FOR MAJOR AND MINOR STRUCTURES:

1. If the operand is a major structure with multiple copies, each copy is unraveled in turn, in order of increasing index. If the operand is a minor structure of a multiple-copy structure, then the copy of the minor structure in each structure copy is unraveled in turn in order of increasing index.
2. The method of unraveling a copy is as follows: each structure terminal on a "branch" connecting back to the given major or minor structure operand is unraveled in turn; the order taken is the order of appearance of the terminals in the structure template.
3. Each structure terminal is unraveled according to the rules given below:

example:

```
STRUCTURE A:
  1 B,
    2 D VECTOR(3),
  1 E INTEGER;
DECLARE A A-STRUCTURE(3);
```

- order of unraveling of B is B_i, i = 1, 2, 3
- order of unraveling of each B_i is C_i , D_i

Figure 5-11 Structure Unraveling Example

RULES FOR OTHER OPERANDS:

1. An operand of any type (integer, scalar, vector, matrix, bit, character, or event) may possess arrayness as described in Section 5.4. Each dimension of arrayness, starting from the leftmost is unraveled in turn, in order of increasing index.
2. Integer, scalar, bit, character, and event types are considered for unraveling purposes as having only one data element.
3. Vector types are unraveled component by component, in order of increasing index.
4. Matrix types are unraveled row by row, in order of increasing index. The components of each row are unraveled in turn in order of increasing index.

example:

```
DECLARE V ARRAY(2,2) VECTOR(3);
```

- order of unraveling of V is $V_{i,*:*}$, i = 1, 2
- order of unraveling of each $V_{i,*:*}$, is $V_{i,j:}$, j = 1, 2
- order of unraveling of each $V_{i,j}$:* is $V_{i,j:k}$, k = 1, 2, 3

(standard HAL/S subscript notation used)

6.0 DATA MANIPULATION AND EXPRESSIONS

An expression is an algorithm used for computing a value. In HAL/S, expressions are formed by combining together operators with operands in a well-defined manner. Operands generally are variables, literals, other expressions, and functions. The type of an expression is the type of its result, which is not necessarily the same as the types of its operands.

In HAL/S, expressions are divided into three major classes according to their usage:

- regular expressions appear in a very large number of contexts throughout the language; e.g., in assignment statements, as arguments to procedures and functions, and in I/O statements. Typical regular expressions are arithmetic, bit, and character. They are collectively denoted by <expression>.
- conditional expressions are used to express combinations of relationships between quantities, and are found in IF statements, and in WHILE and UNTIL phrases. They are denoted by <condition>.
- event expressions are used exclusively in real time programming statements.

6.1 Regular Expressions.

Regular expressions comprise arithmetic expressions, bit expressions, and character expressions, together with a limited form of structure expression. As a generic form, <expression> appears in the assignment statement as the input arguments of procedure and function blocks, and in the WRITE statement.

SYNTAX:

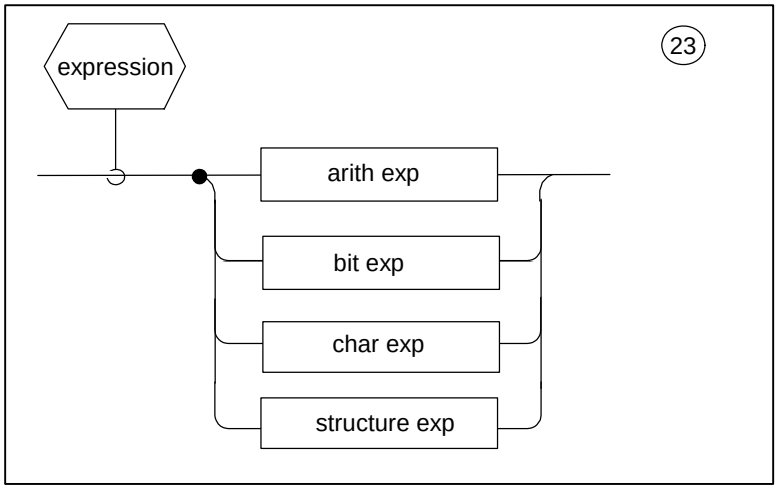


Figure 6-1 expression - #23

Descriptions of <arith exp, <bit exp>, <char exp>, and <structure exp> are given in the following subsections.

6.1.1 Arithmetic Expressions.

Arithmetic expressions include integer, scalar, vector, and matrix expressions. Collectively they are known by the syntactical term <arith exp>.

SYNTAX:

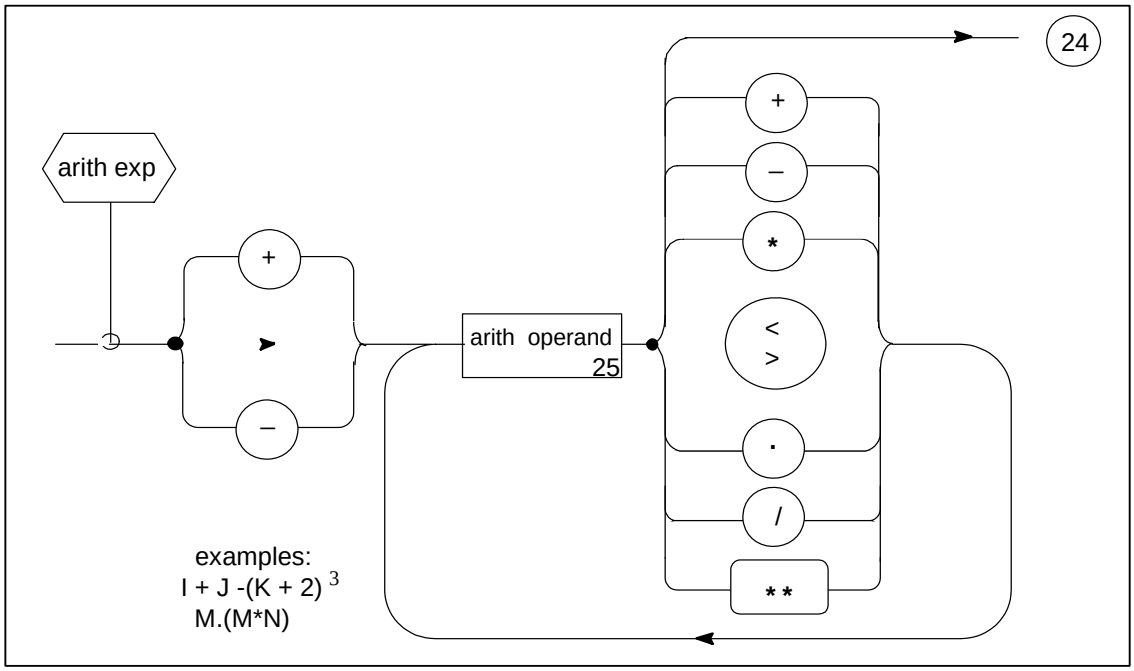


Figure 6-2 arithmetic expression - #24

SEMANTIC RULES:

1. An <arith exp> is a sequence of <arith operand>s separated by infix arithmetic operators, and possibly preceded by a unary plus or minus.
2. The form < > is used to show that the two <arith operand>s are separated by one or more spaces. It signifies a product between the <arith operand>s.
3. The syntax diagram for <arith exp> produces a sequence extensible on the right. Any sequence produced is not necessarily to be considered as evaluated from left to right. The order of evaluation of each operation in the sequence is dictated by operator precedence.
4. Not all types of <arith operand>s are legal in every infix operation. The following table summarizes all possible forms by indicating the result of each legal operation.

OPERANDS		INFIX OPERATOR						
LEFT	RIGHT	+	-	< >	*	•	/	* *
VECTOR	VECTOR	VECTOR	VECTOR	MATRIX[1]	VECTOR[2]	SCALAR[3]		
VECTOR	MATRIX			VECTOR				
MATRIX	VECTOR			VECTOR				
VECTOR	INTEGER			VECTOR[4]			VECTOR[5]	VECTOR[10]
VECTOR	SCALAR			VECTOR[4]			VECTOR[5]	
INTEGER SCALAR	VECTOR			VECTOR[4]				
MATRIX	MATRIX	MATRIX	MATRIX	MATRIX				
MATRIX	INTEGER			MATRIX[4]			MATRIX[5]	MATRIX[6 & 10]
MATRIX	SCALAR			MATRIX[4]			MATRIX[5]	MATRIX[6]
INTEGER SCALAR	MATRIX			MATRIX[4]				
SCALAR	SCALAR	SCALAR	SCALAR	SCALAR			SCALAR	SCALAR[8]
SCALAR	INTEGER	SCALAR	SCALAR	SCALAR			SCALAR	SCALAR[10]
INTEGER	SCALAR	SCALAR	SCALAR	SCALAR			SCALAR	SCALAR[8]
INTEGER	INTEGER	INTEGER	INTEGER	INTEGER			SCALAR[7]	INTEGER[9] SCALAR

Table 6-1 Infix Operators

Notes:

In operations with vector and matrix operands, the sizes of the operands must be compatible with the operation involved, in the usual mathematical sense.

① outer product.

② cross product - valid for 3-vectors only.

③ dot product.

④ every element of the vector or matrix is multiplied by the integer or scalar.

⑤ every element of the vector or matrix is divided by the integer or scalar.

⑥ if the right operand is literally “T” the transpose is indicated. If the right operand is literally “0” the result is an identity matrix. If the right operand is a positive integer number, a repeated product is implied. If the right operand is a negative integer number, repeated product of the inverse is implied. These are the only legal forms.

⑦ the operands are converted to scalar before division.

⑧ the operation is undefined if the value of the left operand is negative, and the value of the right operand is nonintegral.

⑨ the result is a scalar except if the right operand is a non-negative integral compile time constant in which case the result is an integer.

⑩ If both operands are variables, then the integer is not converted to a scalar and the base’s precision determines the precision of the calculation.

5. Except as noted in Rule 4(+), if one operand in an operation is of INTEGER type and the other operand is of SCALAR, VECTOR, or MATRIX type, the INTEGER is converted to a SCALAR before performing the operation. The precision of the INTEGER is maintained (i.e. INTEGER SINGLE is converted to SCALAR SINGLE, INTEGER DOUBLE is converted to SCALAR DOUBLE).

6. If the two operands of an operation are of differing precision, the result is double precision; otherwise, the precision of the result is the precision of the operands. However, when only one operand is a literal (which includes non-aggregate CONSTANTS), the precision of the result is the same as the precision of the non-literal operand. An aggregate is defined in this case as an ARRAY, STRUCTURE, VECTOR, or MATRIX.

PRECEDENCE RULES:

1. The following table summarizes the precedence rules for arithmetic operators:

Operator	Precedence	Operation
	FIRST	
* *	1	Exponeniation
< >	2	Multiplication
*	3	cross-product
.	4	dot-product
/	5	Division
+	6	addition and unary plus
-	6	subtraction and unary minus
	LAST	

Table 6-2 Precedence Rules for Arithmetic Operators

2. If two operations of the same precedence follow each other, then the following rules apply:
   - operators * *, / are evaluated right-to-left;
   - all other operators are evaluated left-to-right;
3. Overriding Rules 1 and 2, the operators <>, *, and . are evaluated so as to minimize the total number of elemental multiplications required. However, this rule does not modify the effective precedence order in cases where it would cause the result to be numerically different, or the operation to be illegal.

An <arith operand> appearing in an <arith exp> has the following form:

SYNTAX:

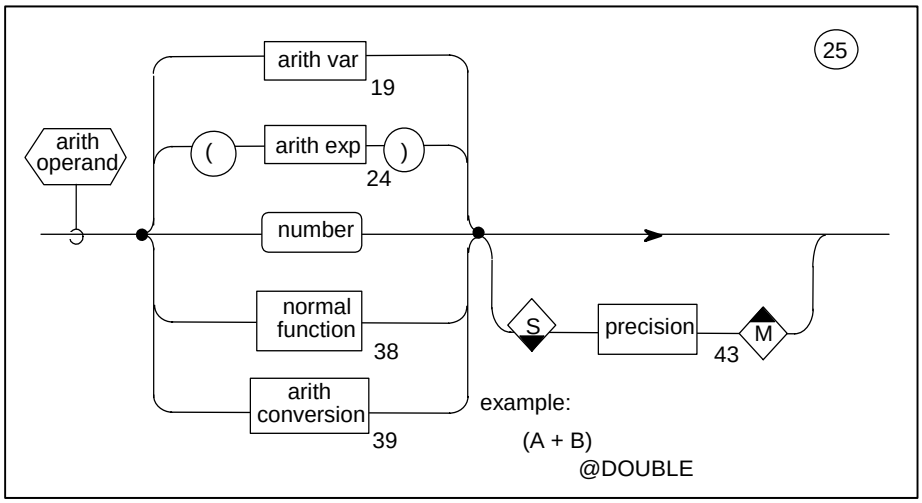


Figure 6-3 arithmetic operand - #25

SEMANTIC RULES:

1. An <arith operand> may be an arithmetic variable, an arithmetic expression enclosed in parentheses, a <normal function> of the appropriate type (see Section 6.4), an <arith conversion> function (see Section 6.5.1), or a literal <number>.
2. The precision of an <arith operand> may be converted by subscripting it with a <precision> specifier (see Section 6.6). If the operand is an <arith var> this is true only if it has no <subscript>[7].
3. Only integer and scalar <arith operand>s may have the form <number>.

7. Since a subscripted <arith var> is an example of an <arith exp>, the <precision> specifier may be applied by first enclosing the <arith exp> in parentheses.

6.1.2 Bit Expressions.

A bit expression is known by the syntactical term <bit exp>.

SYNTAX:

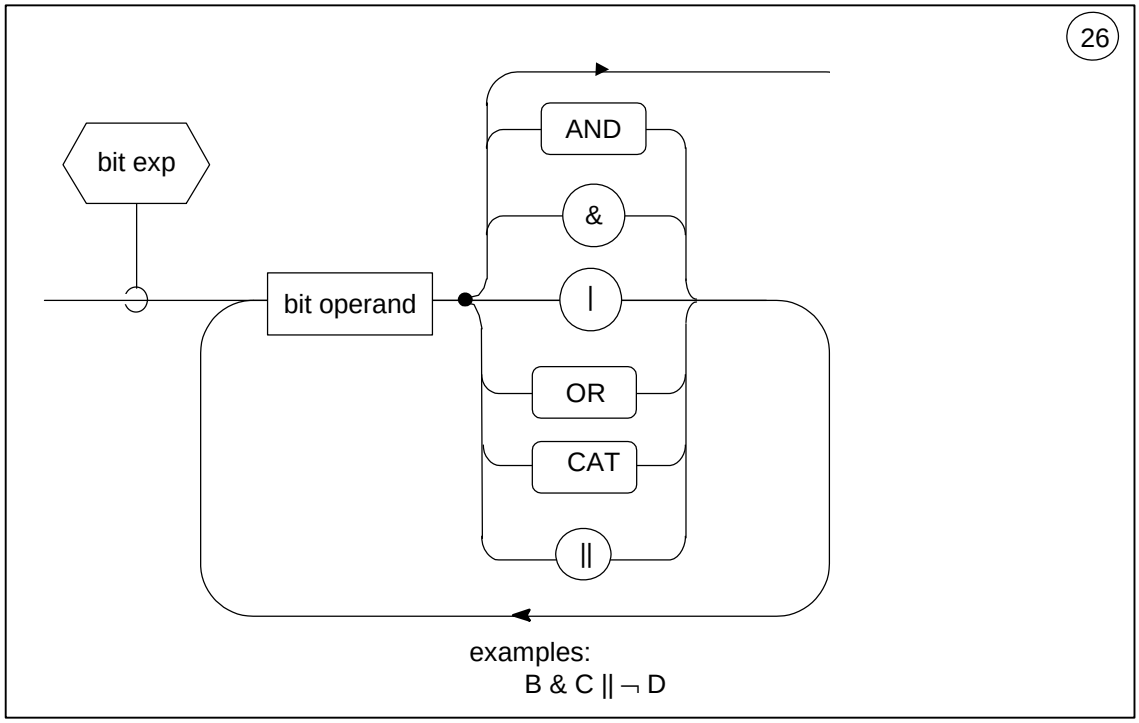


Figure 6-4 bit expression - #26

SEMANTIC RULES:

1. A <bit exp> is a sequence of <bit operand>s separated by bit operators.
2. The syntax diagram for <bit exp> produces a sequence extensible on the right. Any sequence produced is not necessarily to be considered as evaluated from left to right. The order of evaluation of each infix operation is dictated by operator precedence:

Operator	Precedence
	FIRST
CAT, ||	1
AND, &	2
OR, |	3
	LAST

Table 6-3 Precedence Rules for Bit Expressions

   If two operations of the same precedence follow each other, they are evaluated from left to right.
3. The operator CAT (||) denotes catenation of <bit operand>s. The length of the result is the sum of the lengths of the operands.

4. The operators AND (&) and OR (|) denote logical intersection and union, respectively. The shorter of the two <bit operand>s is left padded with binary zeroes to match the length of the longer.

A <bit operand> appearing in a <bit exp> has the following form.

SYNTAX:

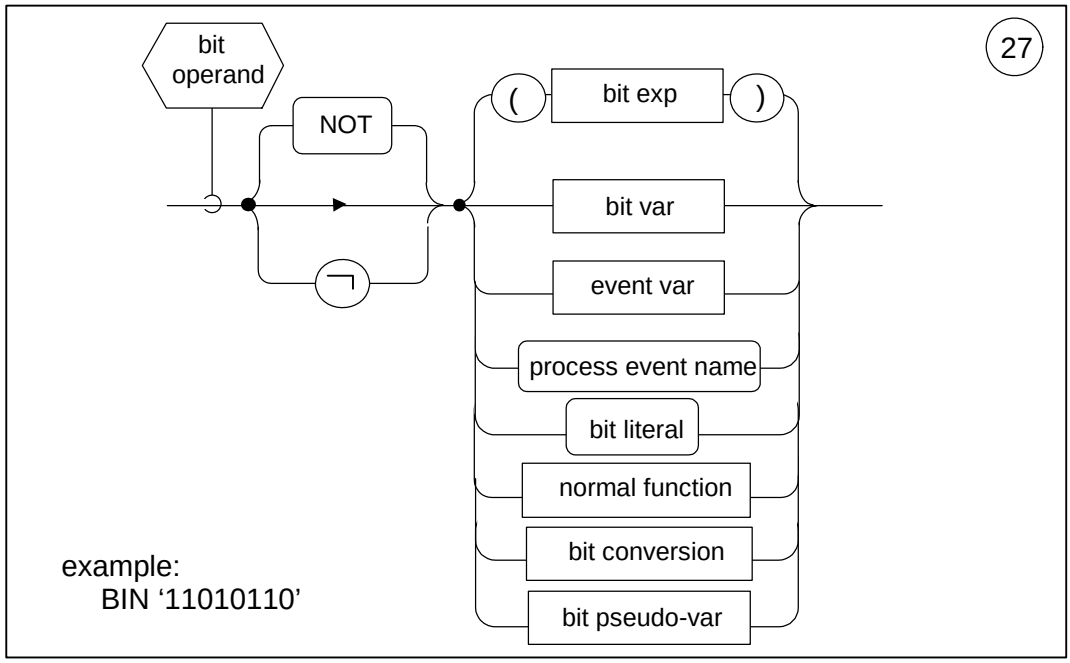


Figure 6-5 bit operand - #27

SEMANTIC RULES:

1. A <bit operand> may be a <bit var>, a <bit exp> enclosed in parentheses, a <bit literal>, a <normal function> of bit type (see Section 6.4), a <bit conversion> function, or a <bit pseudo-var> (see Sections 6.5.3 and 6.5.4).
2. In addition, a <bit operand> may be an <event var> or a <process-event name> (see Section 8.9). Events and process-events are treated as BOOLEAN (1-bit) <bit operand>s.
3. Any form of <bit operand> may be prefaced with the NOT (¬) operator causing its logical complement to be evaluated prior to use within an expression. Note that associating the NOT operation with the <bit operand> syntax achieves an effect similar to placing the NOT operator in the bit expression syntax at the highest level of precedence.

6.1.3 Character Expressions.

A character expression is known by the syntactical term <char exp>.

SYNTAX:

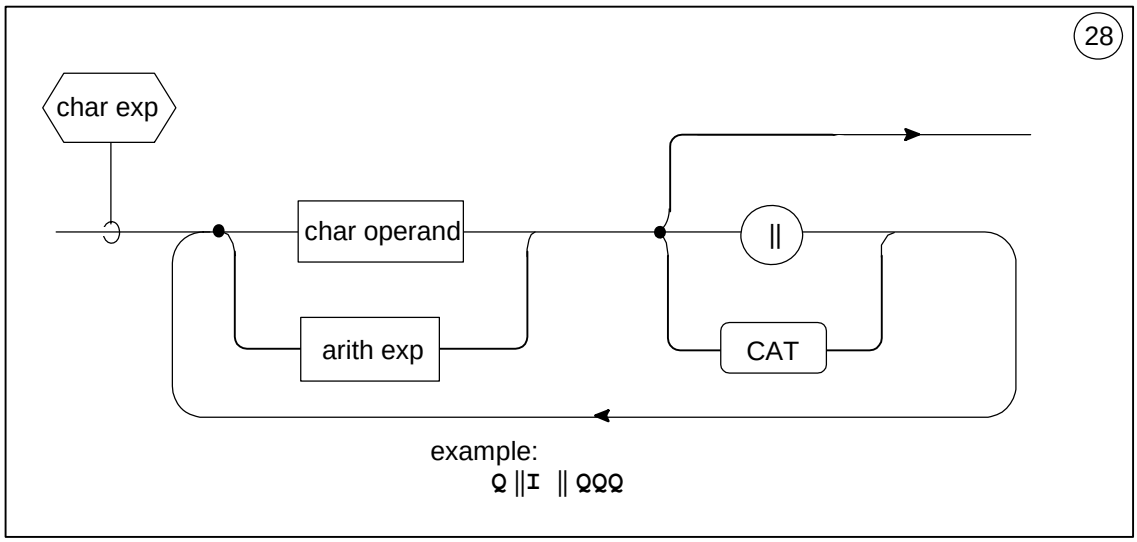


Figure 6-6 character expression - #28

SEMANTIC RULES:

1. A <char exp> is a sequence of operands separated by the catenation operator CAT (||). Each operand may be a <char operand>, or an integer or scalar <arith exp>.
2. The sequence of catenations is evaluated from left to right.
3. Integer and scalar <arith exp>s are converted to character strings according to the standard conversion rules given in Appendix D.

A <char operand> appearing in a <char exp> has the following form.

SYNTAX:

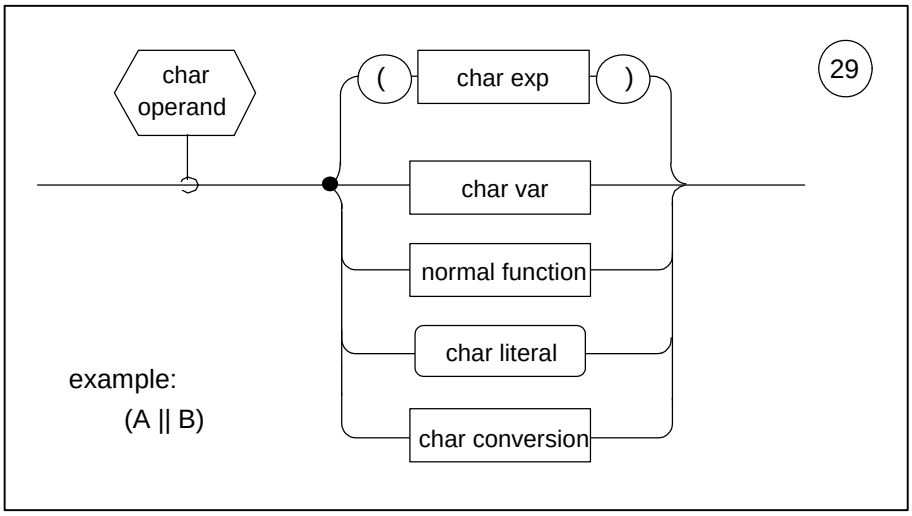


Figure 6-7 character operand - #29

SEMANTIC RULES:

1. A <char operand> may be a character variable, a <char exp> enclosed in parentheses, a <char literal>, a <normal function> of character type (see Section 6.4), or a <char conversion> function (see Section 6.5.3).

6.1.4 Structure Expressions.

Since there are no manipulative expressions for structures, a <structure exp> merely consists of one structure operand.

SYNTAX:

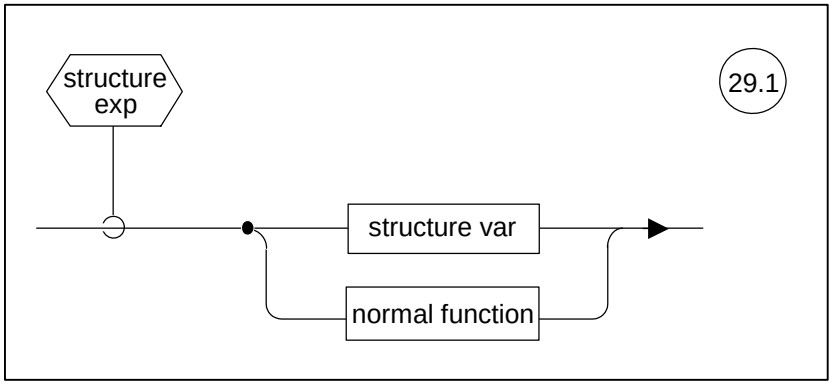


Figure 6-8 structure expression - #29.1

SEMANTIC RULES:

1. A <structure exp> consists of one structure operand which may be either a <structure var>, or a <normal function> of structure type (see Section 6.4).

6.1.5 Array Properties of Expressions.

Any regular expression may have an array property by virtue of possessing one or more arrayed operands. The evaluation of an arrayed regular expression implies element-by-element evaluation of the expression. For any infix operation with an array property the following must be true.

SEMANTIC RULES:

1. If one of the two operands of an infix operation are arrayed, then evaluation of the operation using the unarrayed operand and each element of the arrayed operand is implied. The resulting array has the same dimensions as the arrayed operand.
2. If both of the operands of an infix operation are arrayed, then both operands must have the same dimensions. Evaluation of the operation for each of the corresponding elements of the operands is implied. The resulting array has the same dimensions as the operands.

6.2 Conditional Expressions.

Conditional expressions express combinations of relationships between quantities. The HAL/S representation of a relation between quantities is a <comparison>. <comparison>s are combined with logical operators to form conditional expressions, or <condition>s.

SYNTAX:

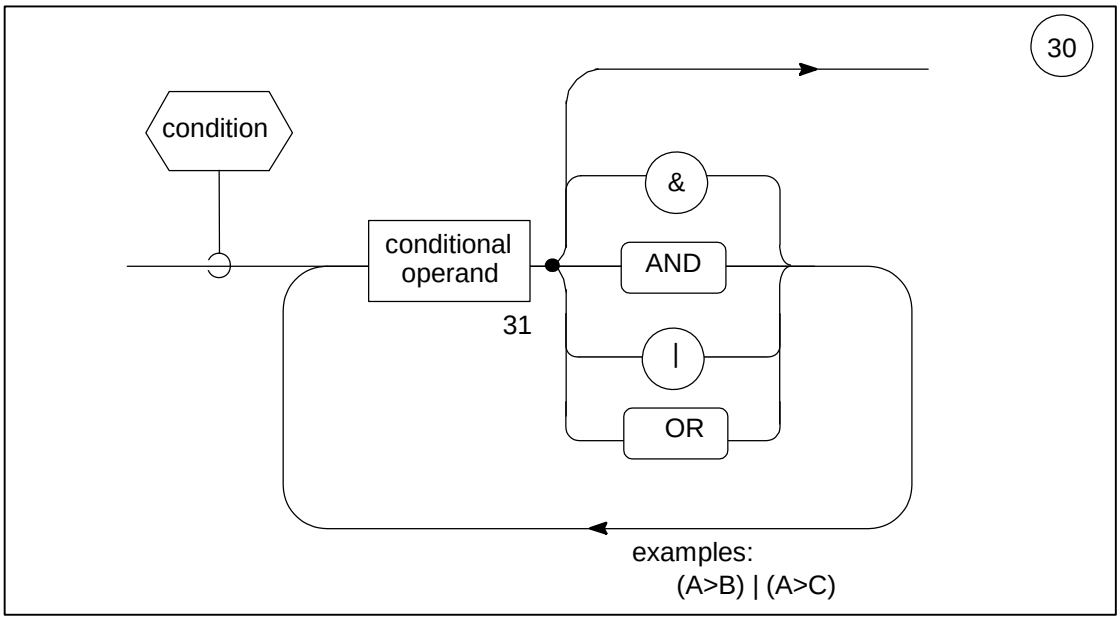


Figure 6-9 conditional expression - #30

SEMANTIC RULES:

1. A conditional expression or <condition> is a sequence of <conditional operand>s separated by logical operators.
2. The syntax diagram for <condition> produces a sequence extensible on the right. Any sequence produced is not necessarily to be considered as evaluated from left to right. The order of evaluation of each infix operation is dictated by operator precedence:

Operator	Precedence
	FIRST
AND, &	1
OR, |	2
	LAST

   If two operations of the same precedence follow each other, they are evaluated from left to right.
3. The operations AND (&) and OR (|) denote logical intersection and union, respectively.

A <conditional operand> appearing in a <condition> has the following form.

SYNTAX:

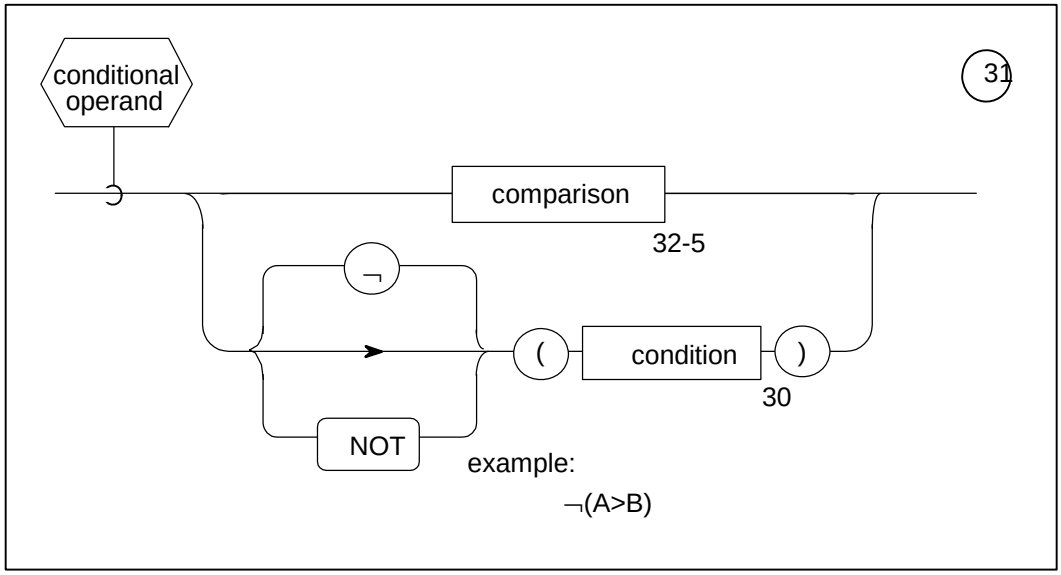


Figure 6-10 conditional operand - #31

SEMANTIC RULES

1. A <conditional operand> is either a <comparison> or a parenthesized <condition>. The latter form may be preceded by the logical NOT (¬) operator.
2. A <comparison> is a relationship between the values of two arithmetic, bit, character, or structure operands. The result of a <comparison> is either TRUE or FALSE, but cannot be used as a boolean operand in a bit expression.

6.2.1 Arithmetic Comparisons.

An arithmetic <comparison> is a comparison between two arithmetic expressions.

SYNTAX:

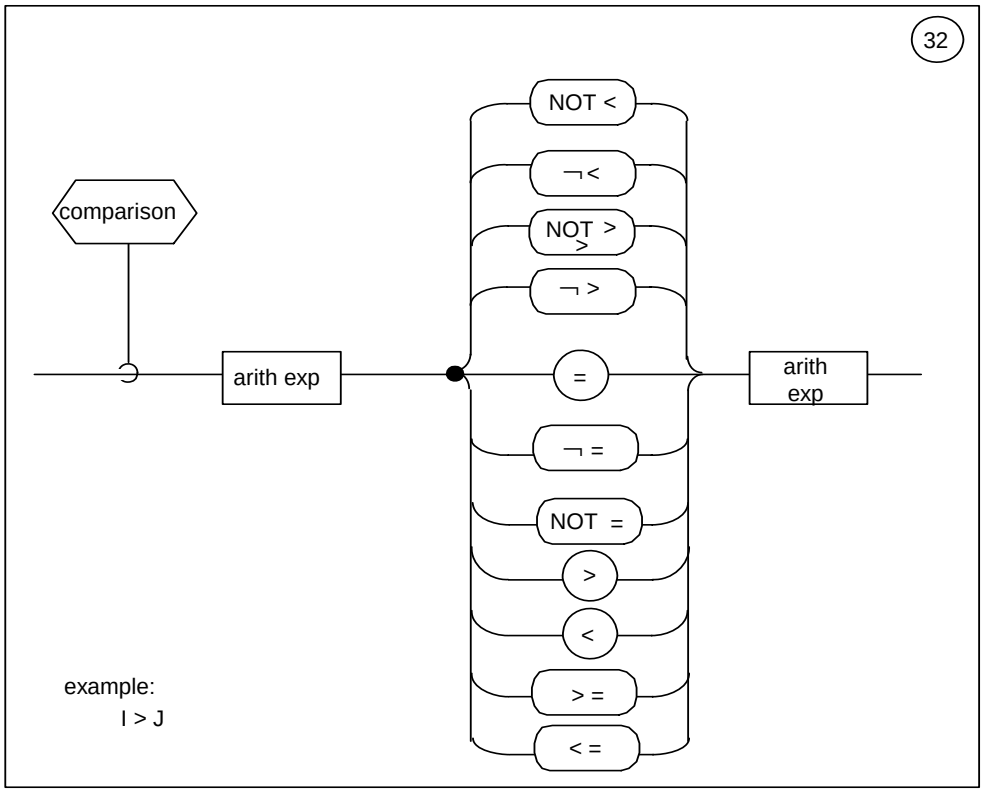


Figure 6-11 arithmetic comparison - #32

SEMANTIC RULES:

1. The types of <arith exp> operand must in general match, with the following exception: in a comparison of mixed integer and scalar operands, the integer operand is converted to a scalar. The precision of the INTEGER is maintained (i.e. INTEGER SINGLE is converted to SCALAR SINGLE, INTEGER DOUBLE is converted to SCALAR DOUBLE).
2. If the precision of the <arith exp> operands are mixed then the single precision operand is converted to double precision.
3. Not all types of <arith exp> are legal for every type of arithmetic comparison. The unshaded boxes in the following table indicate all legal forms.

operands	Operator					
	=	¬= NOT=	>	<	¬> NOT > < =	¬< NOT < > =
VECTOR	√	√				
MATRIX	√	√				
INTEGER SCALAR	√	√	<- - - - - - - no arrays- - - - - - - ->			

4. If the operands are vectors or matrices, the <comparison> is carried out on an element-by-element basis.
   - if the <comparison> operator is= , the result is TRUE only if all the elemental comparisons are TRUE.
   - if the <comparison> operator is NOT= (√=), the result is TRUE if any elemental comparison is TRUE.
5. If one or both of the <arith exp>s are arrayed then only the operators = and NOT= (¬=) are legal, and the result is an arrayed <comparison> (see Section 6.2.5).

6.2.2 Bit Comparisons.

A bit comparison is a comparison between two bit expressions.

SYNTAX:

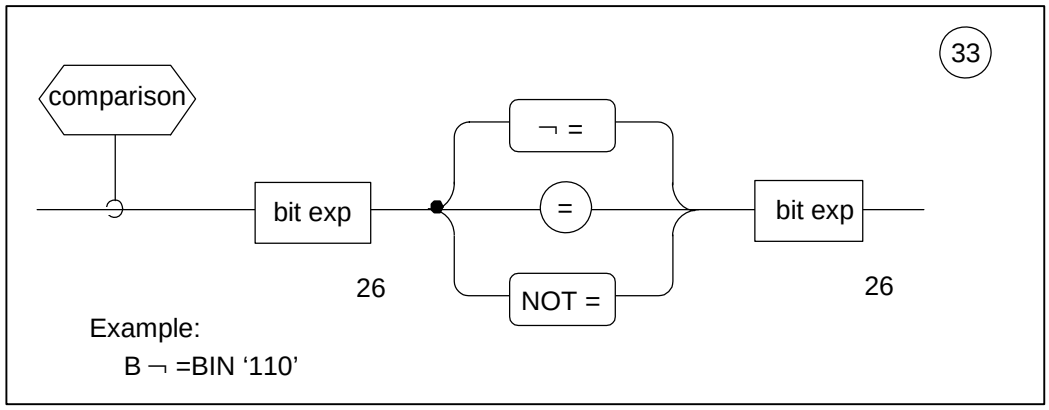


Figure 6-12 bit comparison - #33

SEMANTIC RULES:

1. If the lengths of the operands are the same, their values are equal if and only if they have identical bit patterns.
2. If the lengths of the operands differ, the <bit exp> of shorter length is left padded with binary zeroes to match the length of the longer before comparison takes place.
3. If one or both of the <bit exp>s are arrayed, then the result is an arrayed <comparison> (see Section 6.2.5).

6.2.3 Character Comparisons.

A character comparison is a comparison between two character expressions.

SYNTAX:

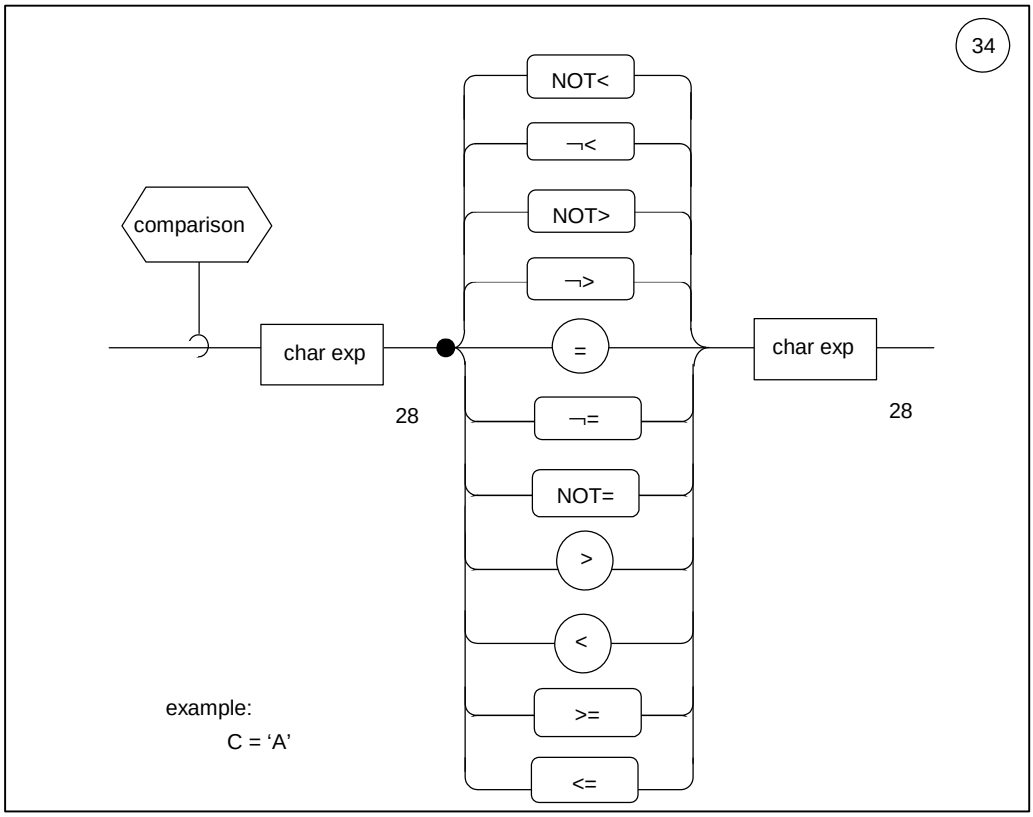


Figure 6-13 character comparison - #34

SEMANTIC RULES:

1. The two strings are compared left-to-right through as many characters as are contained in the shorter string.
2. If a difference in any character is detected, the value of the comparison is determined by the internal character representations of the differing characters (n.b. this is machine dependent).
3. If the shorter string is identical to the longer one truncated to be the same length as the shorter, then it is less than the longer one.
4. If one or both of the <char exp>s are arrayed then the result is an arrayed <comparison> (see Section 6.2.5).

6.2.4 Structure Comparisons.

A structure comparison is a comparison between two structure expressions.

SYNTAX:

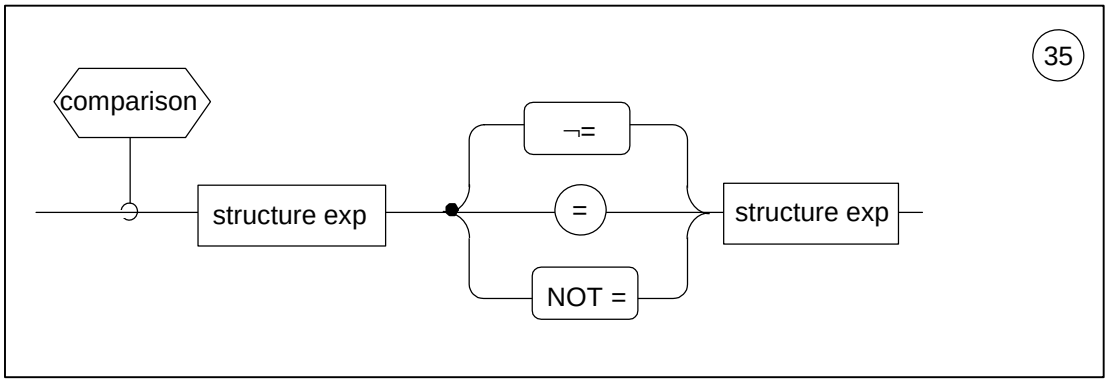


Figure 6-14 structure comparison - #35

SEMANTIC RULES:

1. The tree organizations of both <structure exp>s must be identical in all respects.
2. A multi-copy structure may be compared to either another multi-copy structure or a single copy structure. However, when comparing two multi-copy structures, the number of copies possessed by each <structure exp> must be the same.
3. When comparing a single copy structure to a multi-copy structure, each of the multi-copies is compared to the single copy. If the number of copies is greater than one (for at least one side of the comparison) then the following holds.
   - if the <comparison> operator is =, the result is TRUE only if it is TRUE for all copies.
   - if the <comparison> operator is ¬= (NOT=), the result is TRUE if it is TRUE for at least one pair of corresponding copies.

6.2.5 Comparisons Between Arrayed Operands.

A <comparison> of one of the forms described may have arrayed operands. When one or both of the operands is arrayed, the <comparison> operators are restricted to = and ¬= (NOT=). In any arrayed <comparison>, the following must be true.

SEMANTIC RULES:

1. If one of the two operands of a <comparison> is arrayed, then evaluation of the <comparison> using the unarrayed operand and each element of the arrayed operand is implied.
2. If both of the operands are arrayed, then both operands must have the same array dimensions. Evaluation of the operation for each of the corresponding elements of the operands is implied.

3. The result of an arrayed <comparison> is unarrayed. If the operator is =, then the result is TRUE only if it is TRUE for all elements of the <comparison>. If the operator is ¬= (NOT=), then the result is TRUE if it is TRUE for at least one element of the <comparison>.

6.3 Event Expressions.

Event expressions appear in real time programming statements (see Section 8), and are denoted by the syntactical term <event exp>.

SYNTAX:

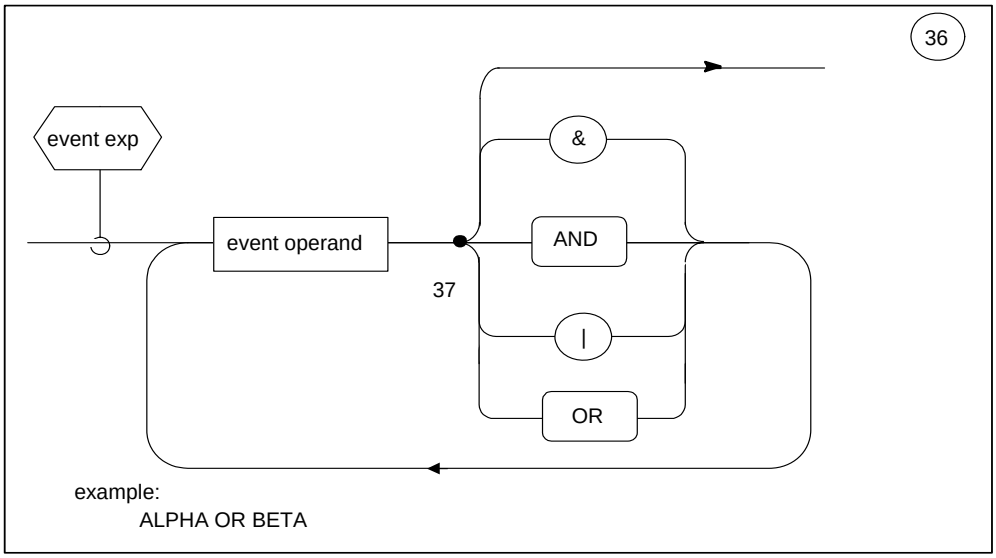


Figure 6-15 event expression - #36

SEMANTIC RULES:

1. An <event exp> is a sequence of <event operand>s separated by a subset of bit operators. An <event exp> may not be arrayed.
2. The syntax diagram for <event exp> produces a sequence extensible on the right. Any sequence produced is not necessarily to be considered as evaluated from left to right. The order of evaluation of each infix operation is dictated by operator precedence:

Operator	Precedence
	FIRST
AND, &	1
OR, |	2
	LAST

Table 6-4 Precedence Rules for Event Expressions

If two operations with the same precedence follow each other, they are evaluated from left-to-right.

3. The operators AND (&) and OR (|) denote logical intersection and union, respectively.

4. The following types of event expressions are the only valid expressions for real time programming statements:
   - single event variable with or without the NOT operator,
   - more than one event variable connected by all OR operators,
   - more than one event variable connected by all AND operators.

An <event operand> appearing in an <event exp> has the following form:

SYNTAX:

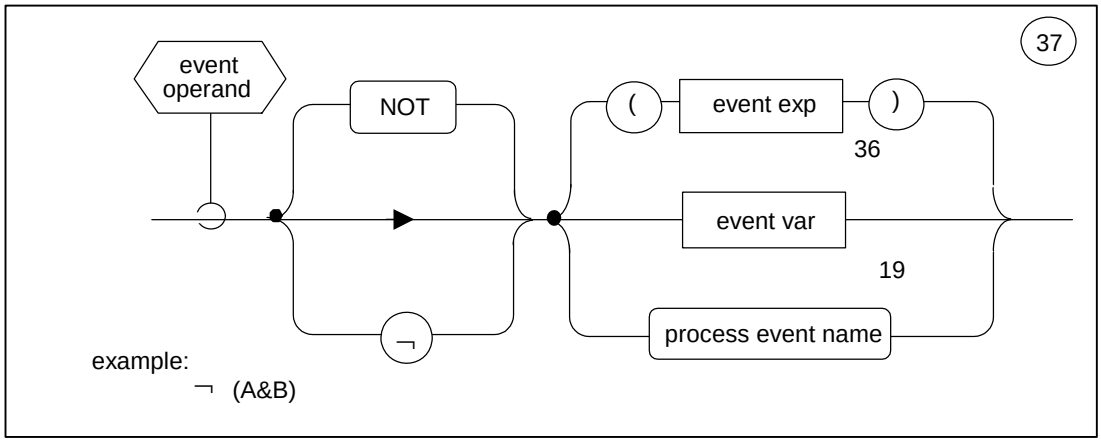


Figure 6-16 event operand - #37

SEMANTIC RULES:

1. An <event operand> may be an event variable, an <event exp> enclosed in parentheses, or a <process-event name>, in which case it is the name of a program or task event.
2. The arrayness of any <event var> must have been removed by suitable subscripting (see Sections 5.3.3 and 5.3.4).
3. The <event operand> may be optionally prefaced by the logical complementing operator NOT (¬).
4. The <process-event name> used as an event operand is that of an external PROGRAM, then a <PROGRAM template> must be included in the compilation unit. The <process-event name> for a TASK block is defined by the occurrence of the TASK block within a PROGRAM block.

6.4 Normal Functions.

Sections 6.1.1 through 6.1.3 have made references to normal functions which may appear as operands in various types of <expression>. The normal function comprises all those functions which are not conversion functions, and fall into two classes:

- “built-in” functions defined as part of the HAL/S language;
- “user-defined” functions defined by the presence of <function block>s in <compilation>s.

The manner of invoking each class of function is essentially the same.

SYNTAX:

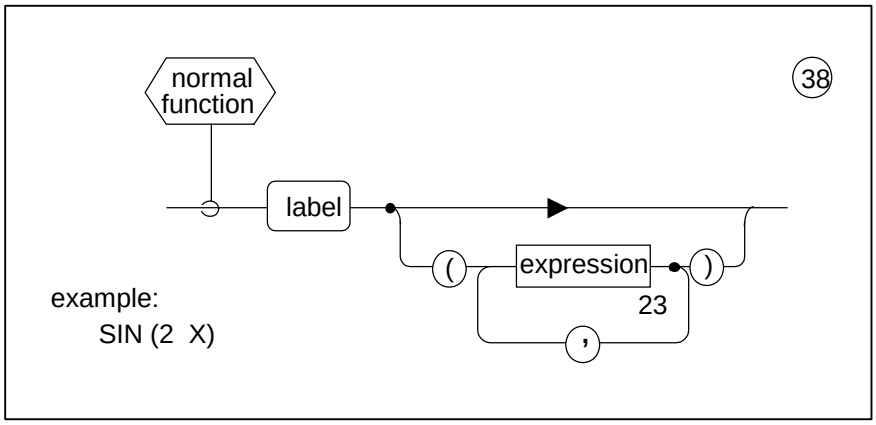


Figure 6-17 normal function - #38

SEMANTIC RULES:

1. <label> invokes execution of a function with name <label>.
2. If <label> is a reserved word which is a built-in function name, then that built-in function is invoked. A list of built-in function names is given in Appendix C.
3. If a <function block> with name <label> appears in such a name scope that <label> is known to the invocation, then that block is invoked.
4. If no such <function block> exists, then the <function block> is assumed to be external to the <compilation> containing the invocation. A <function template> for that <function block> must therefore be present in the <compilation> (see Section 3.6).
5. If a <function block> is declared inside a DO...END group, it may only be invoked by a <normal function> call contained in the same DO...END group.
6. The type of the <normal function> must be appropriate to the type of the <expression> containing it (see Sections 6.1.1 through 6.1.3).
7. Each of the <expression>s in the syntax diagram is an “input argument” of the function invocation. Input arguments are “call-by-reference” or “call-by-value”[8].
8. Each input argument of a <normal function> must match the corresponding input parameter of the function definition[9] exactly in type, dimension, and tree organization, as applicable, except for the following relations:
   - precisions need not match, precision conversions are allowed;
   - the lengths of bit arguments need not match (if the input parameter is not the same length as the input argument, the latter is left truncated or left padded with binary zeroes as necessary. Length mismatch is not allowed for arrayed bit parameters.);
   - CHARACTER arguments must be declared CHARACTER(*);
   - implicit integer to scalar and scalar to integer conversions are allowed;

8. See Section 7.4.
9. The parameter specifications for built-in functions is part of the formal definition given in Appendix C.

- implicit integer and scalar to character conversions are allowed (This conversion is not allowed for arrayed character parameters).

Input arguments may be viewed as being assigned to their respective input parameters on invocation of the function. The rules applicable in the above relaxations thus parallel the relevant assignment rules given in Section 7.3.

9. If the appearance of an invocation of a user-defined function precedes the appearance of its <function block>, the name and type of the function must be declared at the beginning of the containing name scope (see Section 4.6).
10. Special considerations relate to arrayed input arguments to the <normal function>. If the corresponding input parameter is arrayed, then the arrayness must match in all respects. In this case, the function is invoked once. If the corresponding parameter is not arrayed, then the arrayness must match that of the <expression> containing the function. In this case, the <normal function> is invoked once for each array element.
11. The precision of a <normal function> is not necessarily the same as the precision of any of its parameters. The precision of a user-defined <normal function> is defined in the type specification (see Section 4.7).

    The precision for a "built-in" function is defined in Section 5 of the HAL/S-FC Compiler System Specification.

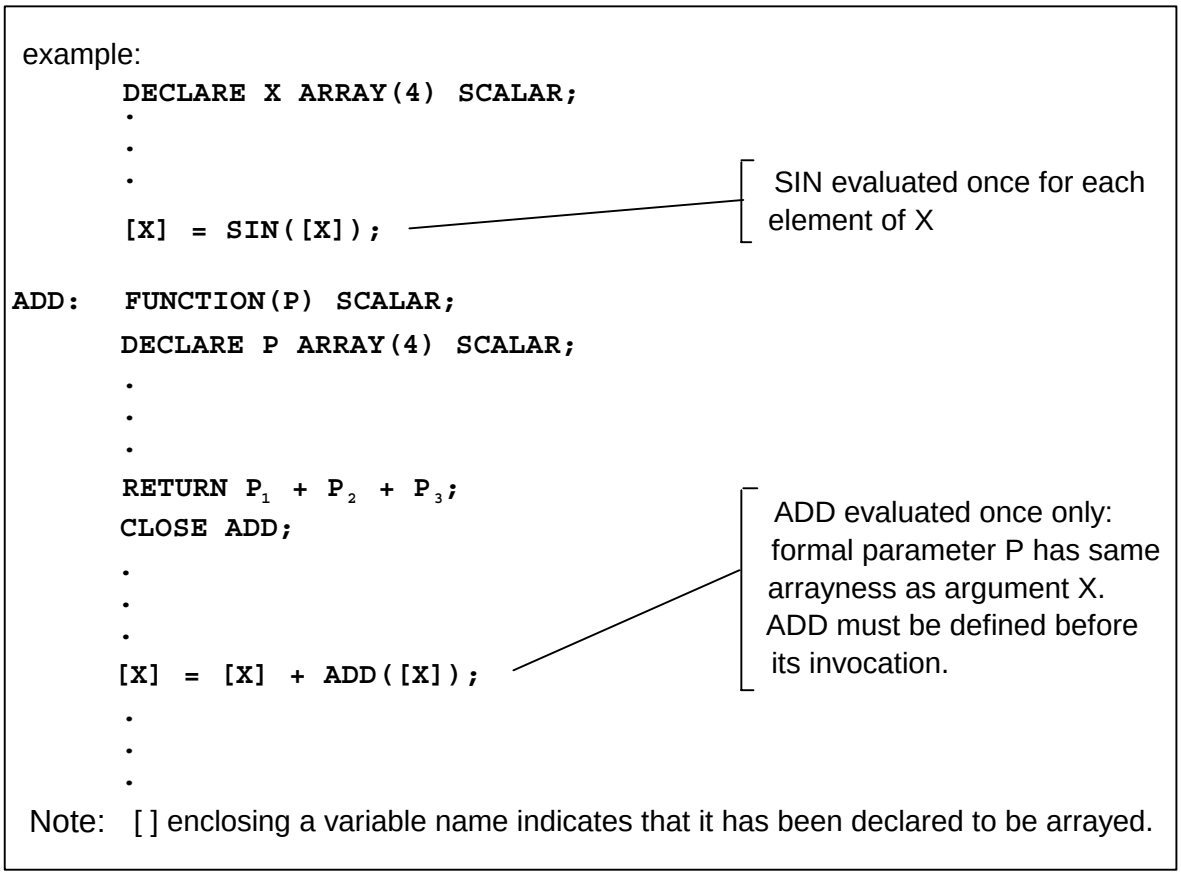


Figure 6-18 Normal Functions Examples

6.5 Explicit Type Conversions.

The limited implicit type conversions offered by HAL/S are described elsewhere in this Specification (see Sections 6.1.1 and 7.3). HAL/S contains a comprehensive set of function- like explicit conversions, some of which also have the property of being able to shape lists of arguments into arrays of arbitrary dimensions. For this reason, conversion functions are sometimes referred to as “shaping functions”. HAL/S contains conversion functions to integer, scalar, vector, matrix, bit, and character types.

6.5.1 Arithmetic Conversion Functions.

Arithmetic conversion functions include conversions to integer, scalar, vector, and matrix types.

SYNTAX:

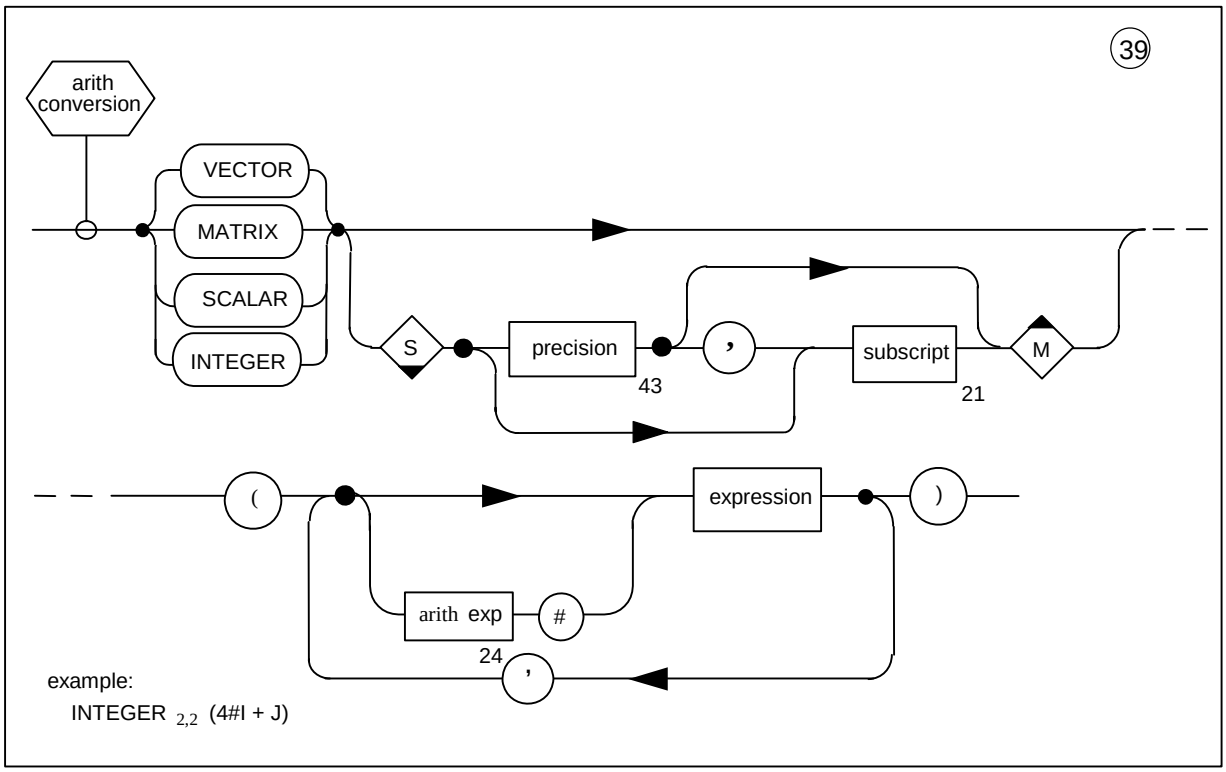


Figure 6-19 arithmetic conversion function - #39

GENERAL SEMANTIC RULES:

1. The keyword INTEGER, SCALAR, VECTOR, and MATRIX gives the result type of the conversion.
2. The conversion keyword is optionally followed by a <precision> specifier giving the precision of the result (see Section 6.6), and by a <subscript> specifying its dimensions.

3. The conversion has one or more <expression>s as arguments. The total number of data elements implied by the argument(s) are shaped according to well-defined rules to generate the result. The data elements in each <expression> are unraveled in their "natural sequence"[10]. The result of doing this for each argument in turn is a single linear string of data elements. This string is then reformed or "reraveled" to generate the result.
4. Any <expression> may be preceded by the phrase <arith exp>#, where <arith exp> is an unarrayed integer or scalar expression computable at compile time (see Appendix F). The value of <arith exp> is rounded to the nearest integer and must be greater than zero. It denotes the number of times the following <expression> is to be used in the generation of the result of the conversion.
5. The nesting of <arith conversion>s is subject to implementation dependent restrictions.

SEMANTIC RULES (INTEGER AND SCALAR):

1. If INTEGER or SCALAR are unsubscripted, and have only one unrepeated argument of integer, scalar, bit, or character type, then if the argument is arrayed, the result of the conversion is identically arrayed.
2. If INTEGER or SCALAR are unsubscripted, and Rule 1 does not apply, then the result of the conversion is a linear (1-dimensional) array whose length is equal to the total number of data elements implied by the argument(s).
3. If INTEGER or SCALAR are subscripted, the form of the <subscript> must be a sequence of <arith exp>s separated by commas. The number of <arith exp>s is the dimensionality of the array produced. Each <arith exp> is an unarrayed integer or scalar expression computable at compile time. Values are rounded to the nearest integer and must be greater than one. They denote the size of each array dimension produced. Their products must therefore match the total number of elements implied by the argument(s) of the conversion.
4. INTEGER and SCALAR may have arguments of any type (subject to general rule 6 above) except structure. Type conversion proceeds according to the standard conversion rules set out in Appendix D.
5. The precision of the result is SINGLE unless forced by the precision of a <precision> specifier.

SEMANTIC RULES (VECTOR AND MATRIX):

1. In the absence of a <subscript>, VECTOR produces a single 3-vector result; MATRIX produces a single 3-by-3 matrix result. The number of data elements implied by the argument(s) must therefore be equal to 3 and 9 respectively.
2. VECTOR and MATRIX cannot produce arrays of vectors and matrices. Consequently, <subscript> may only indicate terminal subscripting.

10. See Section 5.5.

3. In VECTOR, the <subscript> must be an <arith exp>. <arith exp> must be an unarrayed integer or scalar expression computable at compile time (see Appendix F). Its value is rounded to the nearest integer, and must be greater than one. It denotes the length of the vector produced by the conversion. It must therefore match the total number of data elements implied by the argument(s) of the conversion.
4. In MATRIX, the form of the <subscript> must be:

   <arith exp>,<arith exp>

   Each <arith exp> is an unarrayed integer or scalar expression computable at compile time. Values are rounded to the nearest integer, and must be greater than one. They denote the row and column dimensions, respectively, of the matrix produced by the conversion. Their product must therefore match the total number of data elements implied by the argument(s) of the conversion.
5. VECTOR and MATRIX may have arguments of integer, scalar, vector, and matrix types only.
6. The precision of the result is SINGLE unless forced by the presence of a <precision> specifier.

Examples:

```
DECLARE X ARRAY(2,3) SCALAR,
.         V VECTOR(3);
.
.
```

`INTEGER([X])`	result is 2,3 array of integers
`INTEGER([X],[X])`	result is linear 12-array of integers
`SCALAR(V)`	result is linear 3-array of scalars
$\texttt{INTEGER}_{2,6}\texttt{(2\#[X])}$	result is 2,6 array of integers *
`MATRIX(3#V)`	result is 3 by 3 matrix, each row being equal to V
$\texttt{VECTOR}_{6}\texttt{([X])}$	vector of length 6

Note: A variable enclosed in [ ] denotes that it is arrayed.

Figure 6-20 Explicit Conversion Examples

* For example:

Let [X] = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

1. Argument 2# [X] is “first unraveled”, i.e.

   $\begin{bmatrix} 1\,2\,3\,4\,5\,6 & 1\,2\,3\,4\,5\,6 \end{bmatrix}$

   ---------- ----------

2. Linear string is then “reraveled” into 2 X 6 array:

   $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$

6.5.2 The Bit Conversion Function.

Conversion to bit type is carried out by the BIT conversion function.

SYNTAX:

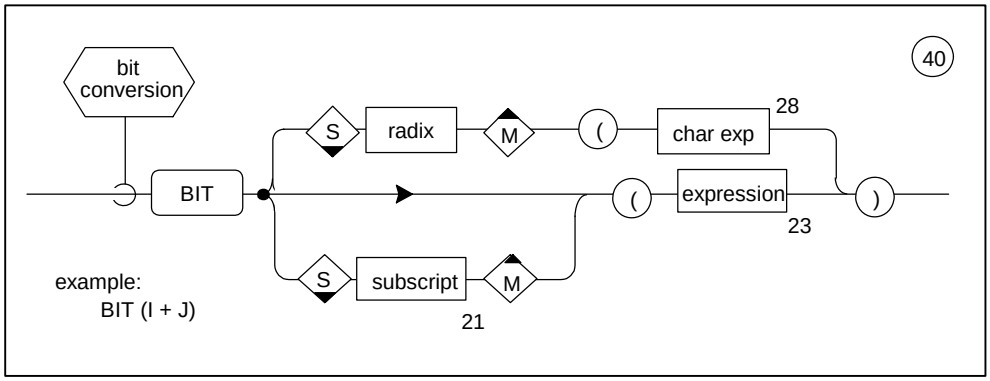


Figure 6-21 bit conversion function - #40

GENERAL SEMANTIC RULES:

1. The keyword BIT denotes conversion to bit type.
2. The conversion has one argument of integer, scalar, bit, or character type. If the argument is arrayed, the result of the conversion is identically arrayed.

SEMANTIC RULES (without <radix>):

1. Conversions of the argument proceed according to the standard conversion rules given in Appendix D. The resulting bit string is of maximum length for the implementation and the significant data is right justified within the word.
   (For an unsubscripted BIT conversion, the resulting bit string is a halfword (for a single precision integer argument), or a bit string of length equal to the argument's length (for a bit type argument)).
2. <subscript> represents component subscripting upon the results of the conversion. <subscript> has the same semantic meaning and restrictions in the current context as it does in the subscripting of bit <variables>s (see Section 5.3.5).

SEMANTIC RULES (with <radix>):

1. The single argument of the <radix> version of the BIT conversion must be a <char exp>. <radix> specifies a radix of conversion, and has one of the following syntactical forms:

   ```
   @HEX      (hexadecimal)
   @DEC      (decimal)
   @OCT      (octal)
   @BIN      (binary)
   ```

2. The <char exp> must consist of a string (or array of strings) of digits legal for the specified <radix>; otherwise, a run-time error occurs. The conversion generates the binary representation of the digit string.
3. During conversion, if the length of the result is too long to be represented in an implementation, left truncation occurs.

Examples:

```
DECLARE X ARRAY(2,3) SCALAR;
    .
    .
    .
```

`BIT([X])`	result is 2,3 array of bit strings
`BIT`$_{1\ TO\ 16}$`([X])`	same as above except that only bits 1 through 16 of each array element are taken
`BIT`$_{@HEX}$`('FACE')`	result is bit pattern of hexadecimal digits represented by argument

Note: A variable enclosed in [ ] denotes that it is arrayed.

6.5.3 The Character Conversion Function.

Conversion to character type is carried out by the CHARACTER conversion function.

SYNTAX:

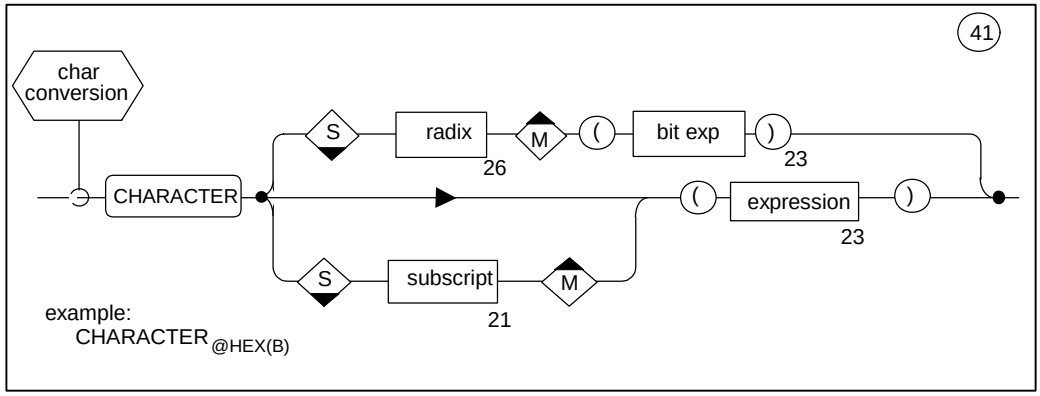


Figure 6-22 bit conversion function - #41

GENERAL SEMANTIC RULES:

1. The keyword CHARACTER denotes conversion to character type.
2. The conversion has one argument of integer, scalar, bit, or character type. If the argument is arrayed, the result of the conversion is identically arrayed.

SEMANTIC RULES (without <radix>):

1. Conversion of the argument proceeds according to the standard conversion rules given in Appendix D.
2. <subscript> represents component subscripting upon the results of the conversion. It has the same semantic meaning and restrictions in the current context as it does in the subscripting of character <variable>s (see Section 5.3.5).

SEMANTIC RULES (with <radix>):

1. The single argument of the <radix> version of the CHARACTER conversion must be a <bit exp>. <radix> specifies a radix of conversion, and has one of the following syntactical forms:

```
@HEX      (hexadecimal)
@DEC      (decimal)
@OCT      (octal)
@BIN      (binary)
```

2. The value of <bit exp> is converted to the representation indicated by the <radix>, left padding the value with binary zeroes as required. The result is a character string consisting of the digits of the representation.

Examples:

```
DECLARE X ARRAY(2,3) SCALAR;
.
.
.
```

CHARACTER([X])	result is 2,3 array of character strings
CHARACTER$_2$([X])	Same as above except that only the second character of each array element is taken
CHARACTER$_{@DEC}$(BIN'101101')	result is decimal representation of the bit pattern of the argument

Note: A variable enclosed in [ ] denotes that it is arrayed.

Figure 6-23 Character Conversion Examples

6.5.4 The SUBBIT pseudo-variable.

The SUBBIT pseudo-variable is a way of making the bit representation of other data types directly accessible without conversion. It may appear in an assignment context (see Section 7.3) as well as part of an <expression>. It is denoted syntactically by <bit pseudo-var>.

SYNTAX:

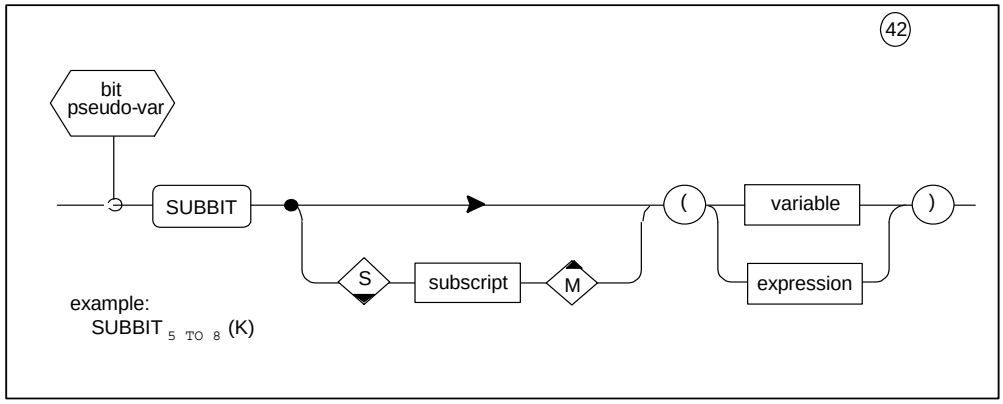


Figure 6-24 SUBBIT pseudo-variable - #42

SEMANTIC RULES:

1. The keyword SUBBIT denotes the pseudo-variable.
2. SUBBIT has one argument only. If it appears in an assignment context, the argument must be a <variable>. If it appears as an operand of a bit expression, the argument must be an <expression>.
3. The argument may be of integer, scalar, bit, or character type, and may optionally be arrayed.
4. The effect of SUBBIT is to make its argument look like an operand of bit type (if the argument is arrayed, then it looks like an arrayed bit operand).
5. <subscript> represents component subscripting upon the pseudo-variable. It has the same semantic meaning as if it were subscripting a bit variable (see Section 5.3.5).
6. The length of the argument in bits may in some implementations be greater than the maximum length of a bit operand. Let the maximum length of a bit operand be *N* bits. If SUBBIT is unsubscripted, only the *N* leftmost bits of the machine representation of the data-type of the argument are visible. If the representation is less than *N*, the number of bits visible is equal to the length of the particular data argument.
7. Partitioning subscripts of SUBBIT may make between 2 and *N* bits from the representation of the argument type visible at any time (i.e., the partition size is $\leq N$.). The partition size must be known at compile time. If the representation is less than the specified partition size, binary zeroes are added on the left.
8. In an assignment context, SUBBIT functions may not be nested within SUBBIT functions. Neither may they appear as assign arguments, or in READ or READALL statements.

Example:

```
DECLARE P SCALAR DOUBLE;
.
.
.
SUBBIT33 TO 64(P)      bits 33 through 64 of the machine representation of P
                       look like a 32-bit variable
                       bits 1 through 32 are invisible.
```

Figure 6-25 SUBBIT Example

6.5.5 Summary of Argument Types.

The asterisks in the following table indicate the legal argument types for each conversion function.

Conversion Function	ARGUMENT TYPE					
	INTEGER	SCALAR	VECTOR	MATRIX	BIT	CHARACTER
INTEGER	*	*	*	*	*	*
SCALAR	*	*	*	*	*	*
VECTOR	*	*	*	*		
MATRIX	*	*	*	*		
BIT	*	*			*	*
BIT with <radix>						*
CHARACTER	*	*			*	*
CHARACTER with <radix>					*	
SUBBIT	*	*			*	*

Table 6-5 Legal Argument Types for Conversion Functions

6.6 Explicit Precision Conversion.

The precision specifier may be used to cause explicit precision conversion of integer, scalar, vector, and matrix data types.

SYNTAX:

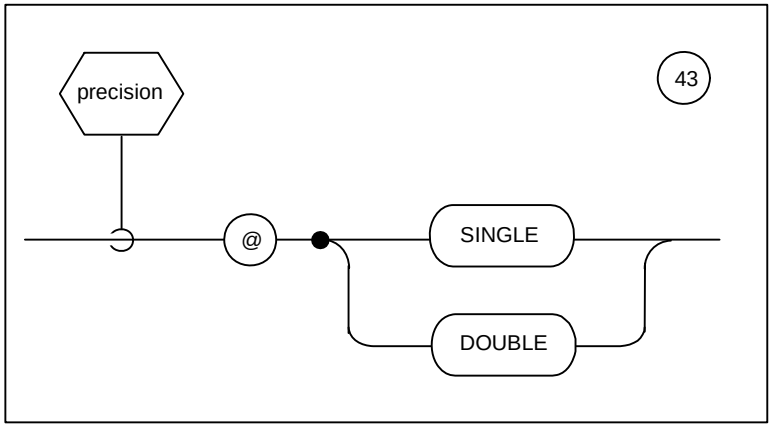


Figure 6-26 precision specifier - #43

SEMANTIC RULES:

1. If <precision> is specified as a subscript to an <arith operand> (see Section 6.1.1), a conversion to the precision specified takes place.
2. If <precision> is specified as a subscript to an <arith conversion> then the result of the conversion is generated with the indicated precision.
3. If referring to integer type, SINGLE implies a halfword, and DOUBLE a fullword. The interpretation is machine dependent.

7.0 EXECUTABLE STATEMENTS

Executable statements are the building blocks of the HAL/S language. They include assignment, flow control, real time programming, error recovery, and input/output statements. Syntactically, a statement of the above type is designated by <statement>. The manner of its integration into the general organization of HAL/S compilation was discussed in Section 3.

7.1 Basic Statements.

All forms of <statement> except the IF statement and certain forms of the ON ERROR statement (see Section 9.1) fall into the category of a <basic statement>.

SYNTAX:

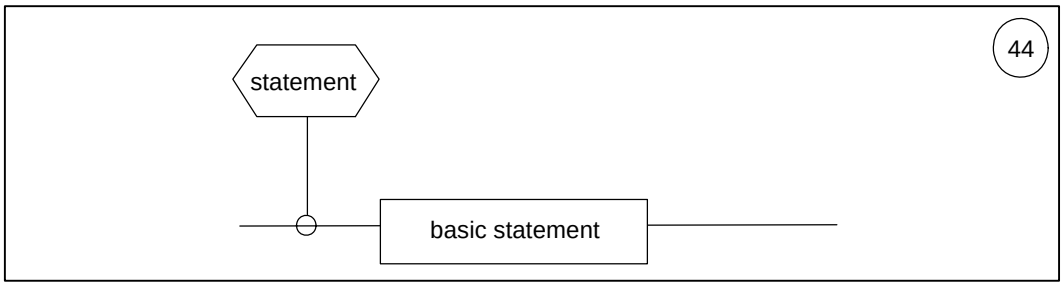


Figure 7-1 basic statement - #44

Any <basic statement>, unless it is imbedded in an IF statement or ON ERROR statement, may optionally be labeled with any number of <label>s. Not all forms of <basic statement> are described in this Section. Real time programming statements are described in Section 8, error recovery statements in Section 9, and input/output statements in Section 10.

7.2 The IF Statement.

The IF statement provides for the conditional execution of segments of HAL/S code.

SYNTAX:

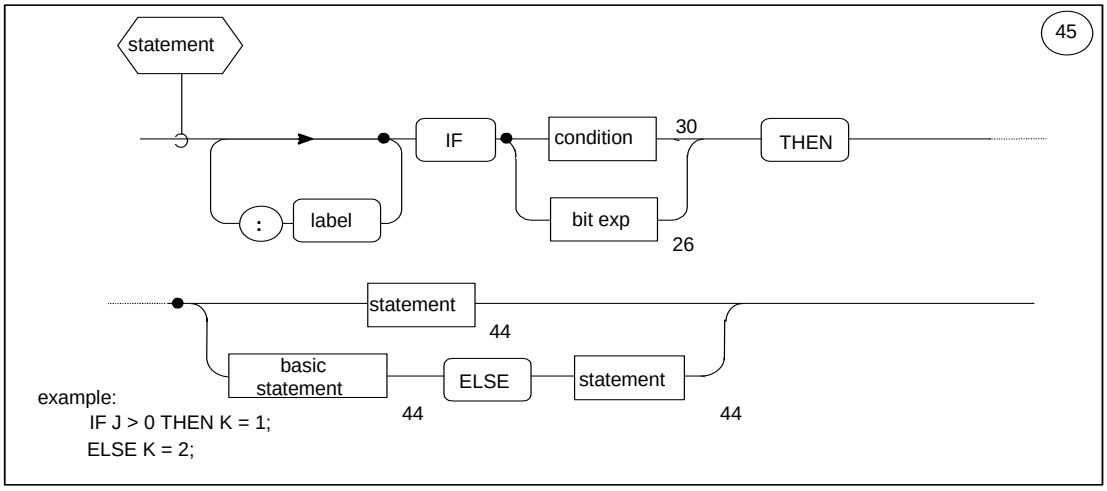


Figure 7-2 IF statement - #45

SEMANTIC RULES:

1. The IF statement, unless it is imbedded in another IF statement or in an ON ERROR statement, may optionally be labeled with any number of <label>s.
2. The option to label the <statement> or <basic statement> of an IF statement is not disallowed. However, such labels may only be referenced by REPEAT or EXIT statements within the (compound) <statement> or <basic statement> thus labeled.
3. If <bit exp> appears in the IF statement, then it must be boolean (i.e., of 1-bit length).
4. If the <condition> or <bit exp> is TRUE, then the <statement> or <basic statement> following the keyword THEN is executed. If <bit exp> is arrayed, then it is considered to be TRUE only if all its array elements are TRUE. Execution then proceeds to the <statement> following the IF statement.
5. If the <condition> or <bit exp> is FALSE then the <statement> or <basic statement> following the keyword THEN is not executed. If the ELSE clause is present, then the <statement> following the keyword ELSE is executed instead, and execution proceeds to the <statement> following the IF statement. If the ELSE clause is absent, execution merely proceeds to the next <statement>.

NOTE: If the ELSE clause is present, a <basic statement> rather than a <statement> precedes the keyword ELSE. A nested IF statement, therefore, cannot appear in this position, thus preventing the well known ‘dangling ELSE’ problem.

7.3 The Assignment Statement.

The assignment statement is used to change the current value of a <variable> or list of <variable>s to that of an expression evaluated in the statement.

SYNTAX:

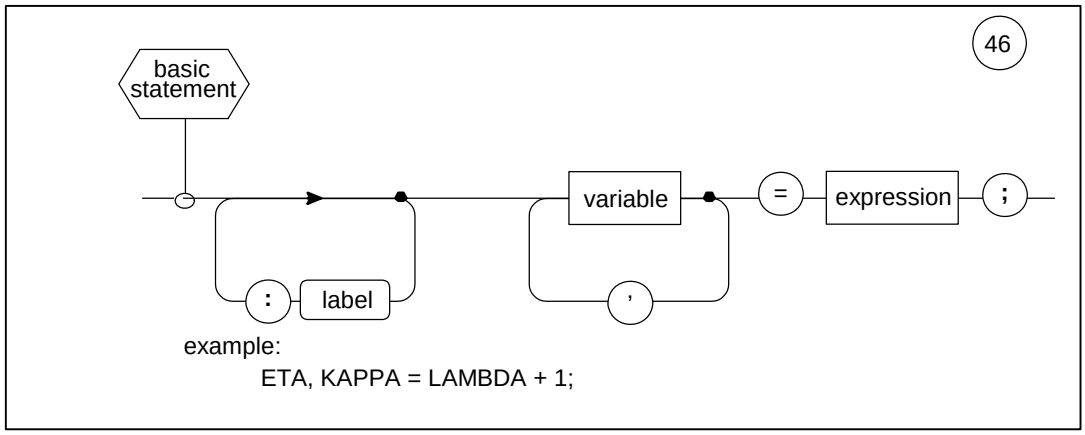


Figure 7-3 assignment statement - #46

GENERAL SEMANTIC RULES:

1. <variable> may not be an event variable or an input parameter of a procedure or function block.
2. The effective order of execution of an assignment statement is as follows:
   - any subscript expressions on the left-hand side are evaluated;
   - the right-hand <expression> is evaluated;
   - the values of the left-hand side <variable>s are changed.
3. If the <expression> on the right-hand side is arrayed, then all the <variable>s on the left-hand side must be arrayed. The number of dimensions of arrayness on each side must be the same, and corresponding dimensions on either side must match in size.
4. If the <expression> on the right-hand side is not arrayed, then it is still possible for one or more <variable>s on the left-hand side to be arrayed. If more than one <variable> is arrayed, the arrayness must match in the sense of General Semantic Rule 3, above. The single unarrayed value will be assigned to every element of arrayed targets.
5. Generally, the type of <expression> must match the types of the <variable>s on the left-hand side. Specific exceptions to this rule are listed below. The type of an assignment is taken to be the same as the type of the <variable> whose value is being changed.

SEMANTIC RULES (integer and scalar assignments):

1. The following implicit type conversions are allowed during assignment:
   - assignment of an integer <expression> to a scalar <variable> is allowed;
   - assignment of a scalar <expression> to an integer <variable> is allowed.
2. If the left- and right-hand sides of a scalar assignment have differing precisions, precision conversion is freely allowed. Conversion from DOUBLE to SINGLE precision implies truncation of an implementation dependent number of binary digits from exponent, mantissa, or both.

SEMANTIC RULES (vector and matrix assignments):

1. The <expression> must normally be a vector or matrix expression with the same type and dimensionality as the <variable>s on the left-hand side. One relaxation of this rule is permitted. Matrix or vector <variable>s may be set null by specifying literal zero for the <expression>. In this case only, both matrices and vectors of any dimension(s) may appear mixed in the list of <variable>s.
2. If the left- and right-hand sides of an assignment have differing precisions, precision conversion is freely allowed, according to the semantic rules for scalar assignments given above.

SEMANTIC RULES (bit assignments):

1. If the length of the bit <expression> is unequal to that of the left-hand side bit <variable>, then the result of the <expression> is left-truncated if it is too long, or left-padded with binary zeros if it is too short.
2. The effect of a left-hand side <variable> being a <bit pseudo-var> is described in Section 6.5.4.

SEMANTIC RULES (character assignments):

1. Assignment of an integer or scalar <expression> to a character <variable> is allowed. During assignment, the integer or scalar value is converted to a character string according to the conversion formats given in Appendix D.
2. If <variable> is a character variable with no component subscripting, then:
   - If the length of the <expression> is greater than the declared maximum length of the <variable>, the <expression> is right-truncated to that length. The <variable> takes on its maximum length.
   - If the length of the <expression> is not greater than the declared maximum length of the variable, then <variable> takes on the length of the <expression>.
3. If <variable> is a character variable with component subscripting, then:
   - If the length of the <expression> is greater than the length implied by the component subscript, then it is right-truncated to the implied length.
   - If the length of the <expression> is less than the length implied by the component subscript, then it is right-padded with blanks to the implied length.
   - After assignment the <variable> takes on the length implied by the upper index of the component subscript, or retains its original length, which ever is the greater. If the upper index of the subscript implies a length greater than the declared maximum for that <variable>, right-truncation to the maximum length occurs.
   - If the lower index is greater than the length of the <variable> before assignment, then the intervening gap is filled with blanks.

SEMANTIC RULES (structure assignments):

1. <expression> can only be a <structure exp>. The tree organization of the structure operands on both sides of the assignment must match exactly in all respects. The sense in which tree organizations of two structures are said to match is described in Section 4.3.

7.4 The CALL Statement.

The CALL statement is used to invoke execution of a procedure. The PROCEDURE block may be in the same <compilation> as the CALL statement or external to it.

SYNTAX:

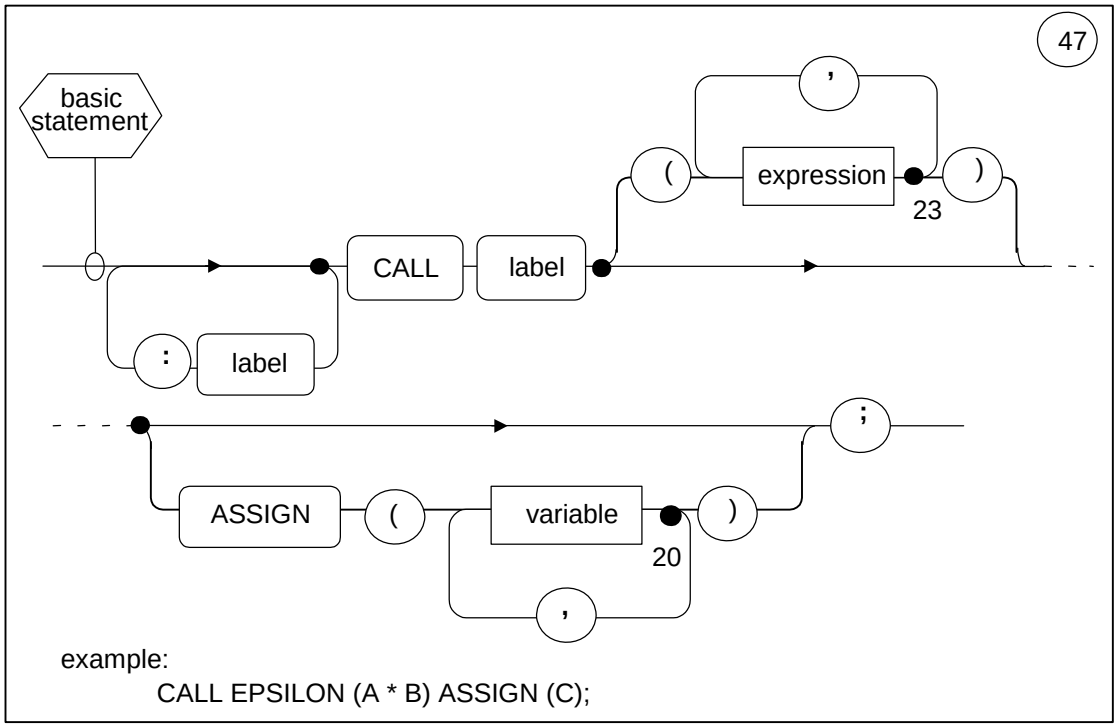


Figure 7-4 CALL statement - #47

SEMANTIC RULES:

1. CALL <label> invokes execution of a procedure with name <label>.
2. If a <procedure block> with name <label> appears in such a name scope that <label> is known to the CALL statement, then CALL <label> invokes that block.
3. If a <procedure block> is declared inside a DO…END group, it may only be invoked by a CALL statement contained in the same DO…END group.
4. If no such <procedure block> exists, then the <procedure block> is assumed to be external to the <compilation> containing the CALL statement. A <procedure template> for that <procedure block> must therefore be present in the <compilation> (see Section 3.6).
5. Each of the <expression>s is an “input argument” of the procedure call.
6. Each of the <variable>s is an “assign argument” of the procedure call. Only assign arguments may have their values changed by the procedure. If <variable> is subscripted, it must be restricted in form to the following:
   - No component subscripting for bit and character types.
   - If component subscripting is present, <variable> must be subscripted so as to yield a single (unarrayed) element of the <variable>.
   - If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

7. Assign arguments are “call-by-reference”. Input arguments are either “call-by-reference” or “call-by-value”[11].
8. Each assign argument must match its corresponding procedure block assign parameter exactly in type, precision, dimension, arrayness, structure tree organization, and REMOTE[12] attribute, as applicable. CHARACTER lengths are an exception; they must be declared CHARACTER(*). The reason is that character types are of varying length and the actual length is available at execution. If an assignment argument has the LOCK attribute, then the following must apply:
   - If it is of lock group N, then the corresponding assign parameter must be of lock group N, or *.
   - If it is of lock group *, then the corresponding parameter must also be of group *.
9. Bit type identifiers (non-NAME, non-arrayed) may not be used as assign arguments of a CALL statement when they are part of structure variables and have DENSE attributes. All other types of structure terminals with the DENSE attribute may be used as ASSIGN arguments. See Sections 4.3 and 4.5 for further explanation of the DENSE attribute. Note, however, that an entire structure with the DENSE attribute may be passed provided that template matching rules are observed.
10. For input arguments, the following relaxation of rules 8 and 9 are permitted:
    - precisions need not match;
    - lengths of bit arguments need not match (If the input parameter is not the same length as the input argument, the latter is left truncated or left padded with binary zeros as necessary. Length mismatch is not allowed for arrayed bit parameters.);
    - CHARACTER arguments must be declared CHARACTER(*);
    - implicit integer to scalar and scalar to integer conversions are allowed;
    - implicit integer and scalar to character conversions are allowed (This conversion is not allowed for arrayed CHARACTER parameters.);
    - matching of the attributes DENSE and REMOTE is not required. (The REMOTE keyword is ignored on non-NAME input parameters.)

    Input arguments may be viewed as being assigned to their respective input parameters on invocation of the procedure. The rules applicable in the above relaxations thus parallel the relevant assignment rules given in Section 7.3.
11. If an assign argument is a structure terminal or a minor structure node (but not if it is a major structure) and if the structure possesses multiple copies, then the number of copies must be reduced to one by subscripting.

11. In this context “call-by-reference” means the arguments are pointed to directly. “Call-by-value” means the value of an input argument, at the invocation of a procedure, is made available to the procedure.
12. Non-REMOTE arguments may be passed into non-NAME REMOTE parameters.

Example:

```
STRUCTURE Z:
   1 A,
     2 C CHARACTER(80),
     2 B VECTOR,
   1 D INTEGER;
DECLARE ZZ Z-STRUCTURE(20);
.
.
.
.
CALL X ASSIGN (ZZ, ZZ.A, ZZ.A.B, ZZ.A1)
               ↑   \_____________/   ↑
             legal     illegal     legal
```

Figure 7-5 CALL ASSIGN Example

7.5 The RETURN Statement.

The RETURN statement is used to cause return of execution from a TASK, PROGRAM, PROCEDURE, or FUNCTION block. In the case of the FUNCTION block it also specifies an expression whose value is to be returned.

SYNTAX:

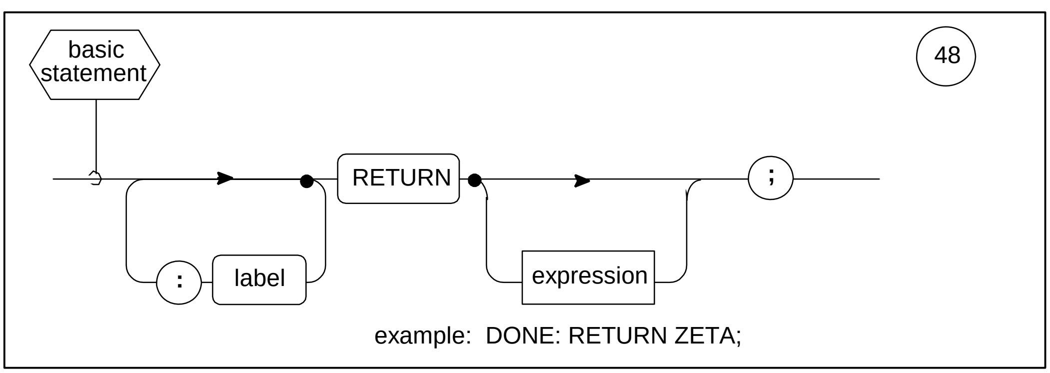


Figure 7-6 RETURN statement - #48

GENERAL SEMANTIC RULES:

1. The effect of the RETURN statement is to cause normal exit (return of execution) from a TASK, PROGRAM, PROCEDURE, or FUNCTION block (also see the CLOSE statement, Section 3.7.4).
2. <expression> may only appear in a RETURN statement of a <function>. Its value is the returned value of the function, and is evaluated prior to returning.

3. <expression> must match the function definition in type and dimension, with the following exceptions:
   - the lengths of bit expressions need not match;
   - the lengths of character expressions need not match;
   - implicit integer to scalar and scalar to integer conversions are allowed;
   - implicit integer and scalar to character conversions are allowed.

   The return of the function values may be viewed as the assignment of the <expression> to the function name. The rules applicable in the above exceptions thus parallel the relevant assignment rules given in Section 7.3.
4. <expression> must always appear in RETURN statements of <function block>s. Execution must always end on logically reaching a RETURN statement of such a block, and not by logically reaching the delimiting CLOSE statement.

7.6 The DO…END Statement Group.

The DO…END statement group is a way of grouping a sequence of<statement>s together so that they collectively look like a single <basic statement>. Additionally, some forms of DO…END group provide a means of executing a sequence of <statement>s either iteratively, or conditionally, or both.

SYNTAX:

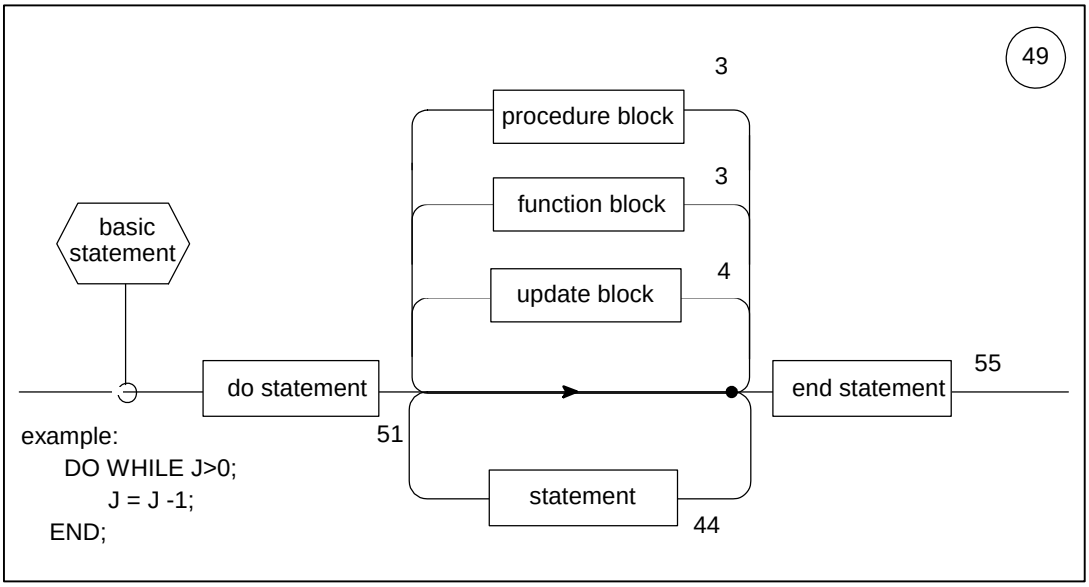


Figure 7-7 DO…END statement group - #49

The DO…END statement group is opened with a <do statement> and closed with an <end statement>. In between may appear any number of <statement>s interspersed as required with FUNCTION, PROCEDURE, or UPDATE blocks. The form of the <do statement> determines how the <statement>s within the group are executed.

7.6.1 The Simple DO Statement.

The simple DO statement merely indicates that the following sequence of <statement>s comprising the group is to be viewed as a single <basic statement>. The sequence is executed only once.

SYNTAX:

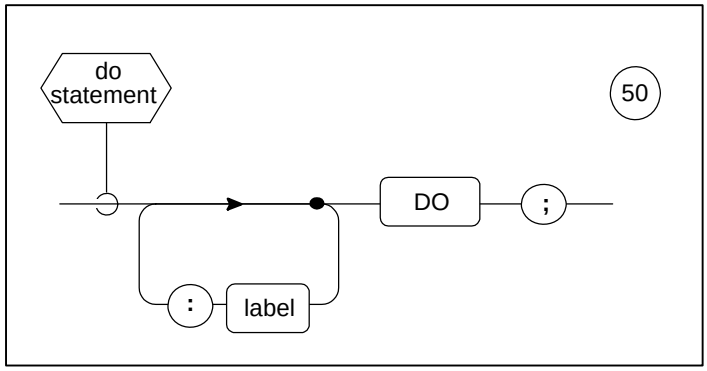


Figure 7-8 simple DO statement - #50

7.6.2 The DO CASEStatement.

The DO CASE statement indicates that in the following sequence of <statement>s comprising the group, only one specified <statement> is to be executed.

SYNTAX:

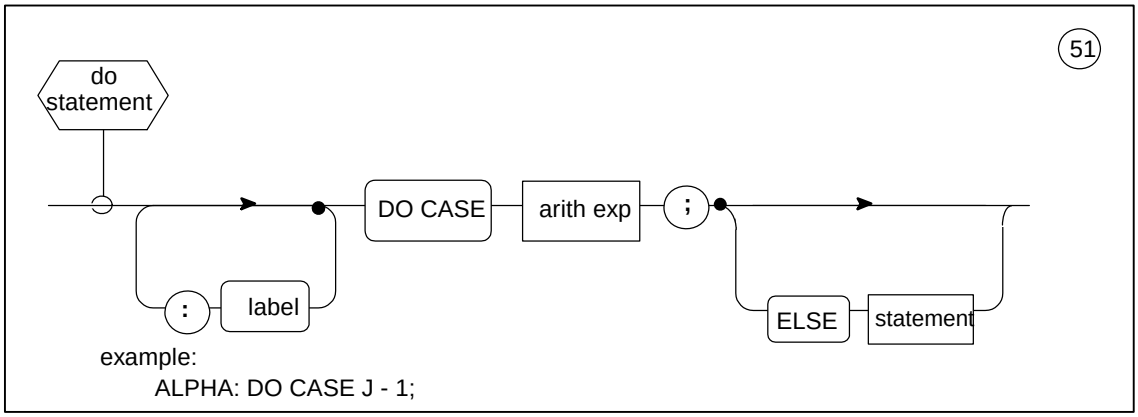


Figure 7-9 DO CASE statement - #51

SEMANTIC RULES:

1. <arith exp> is any unarrayed integer or scalar expression. The value of a scalar expression is rounded to the nearest integer before use.
2. Let the value of <arith exp> be denoted by *K*. If *K* is greater than zero, but not greater than the number of <statement>s in the group, then the K^{th} <statement> of the group is executed.

3. If the value of *K* is outside the range defined in Rule 2, and no ELSE clause appears in the DO CASE statement, then an error condition exists. The result of such an error is implementation dependent.
4. If the value of *K* is outside the range defined in Rule 2, but an ELSE clause does appear, the <statement> following the ELSE keyword is executed instead of one of those in the group. The option to label <statement> is not disallowed. However, such labels may only be referenced by EXIT or REPEAT statements within the (compound) <statement> thus labeled.
5. The presence of any code block definition in the group of <statement>s does not change the *K*-indexing of the <statement>s except for UPDATE blocks (which are considered as single statements).

7.6.3 The DO WHILE and UNTIL Statements.

The DO WHILE and UNTIL statements cause repeated execution of the sequence of <statement>s in a group until some condition is satisfied.

SYNTAX:

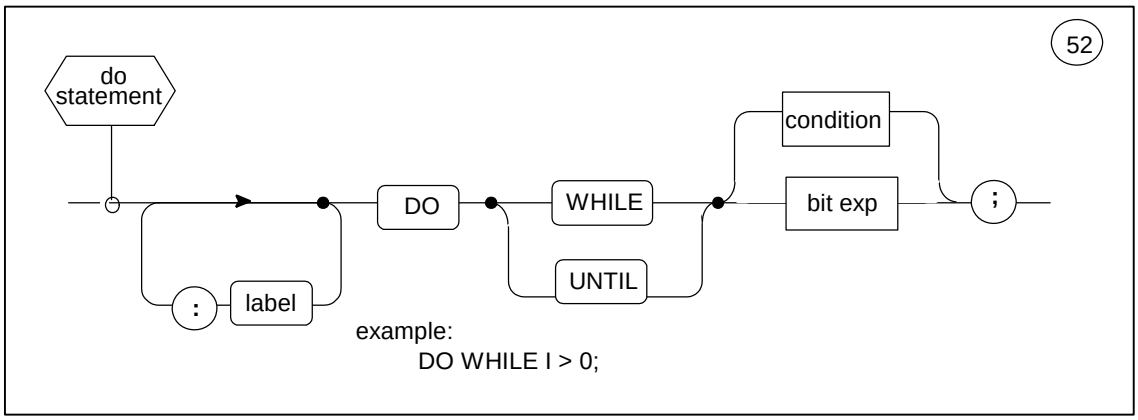


Figure 7-10 DO WHILE and UNTIL statements - #52

SEMANTIC RULES:

1. There is no semantic restriction of <condition>. <bit exp> must be boolean and unarrayed (i.e., of 1-bit length). The <condition> or <bit exp> is reevaluated every time the group of <statement>s is executed.
2. In the DO WHILE version, the group of <statement>s is repeatedly executed until the value of <condition> or <bit exp> becomes FALSE. The value is tested at the beginning of each cycle of execution. This implies that if <condition> or <bit exp> is initially FALSE the group of <statement>s is not executed at all.
3. In the DO UNTIL version, the group of <statement>s is repeatedly executed until the value of the <condition> or <bit exp> becomes TRUE. The value is not tested before the first cycle of execution. On the second and all subsequent cycles of execution, the value is tested at the beginning of each cycle. Use of the UNTIL version therefore guarantees at least one cycle of execution.

7.6.4 The Discrete DO FOR Statement.

The discrete DO FOR statement causes execution of the sequence of <statement>s in a group once for each of a list of values of a “loop variable”. The presence of a WHILE or UNTIL clause can be used to cause such execution to be dependent on some condition being satisfied.

SYNTAX:

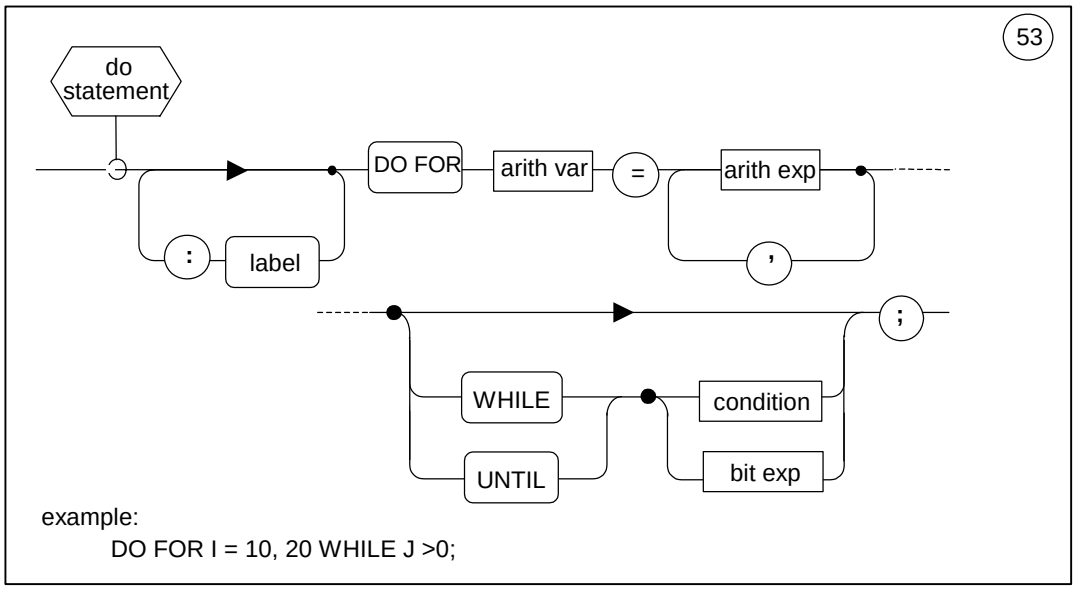


Figure 7-11 discrete DO FOR statement - #53

SEMANTIC RULES:

1. <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable. The initial loop variable, determined after all required subscripting and NAME dereferencing, is used throughout.
2. The maximum number of times of execution of the group of <statement>s is the number of <arith exp>s in the assignment list.
3. <arith exp> is an unarrayed INTEGER or SCALAR expression.
4. At the beginning of each cycle of execution of the group the next <arith exp> in the list (starting from the leftmost) is evaluated and assigned to the loop variable. The assignment follows the relevant assignment statement rules given in Section 7.3.
5. Use of the WHILE or UNTIL clause causes continuation of cycling of execution to be dependent on the value of <condition> or <bit exp>.
6. There is no semantic restriction or <condition>. <bit exp> must be boolean and unarrayed (i.e., of 1-bit length). The <condition> or <bit exp> is reevaluated every time the group of <statement>s is executed.

7. If the WHILE clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes FALSE. The value is tested at the beginning of each cycle of execution after the assignment of the loop variable. This implies that if <condition> or <bit exp> is FALSE prior to the first cycle of execution of the group, then the group will not be executed at all.
8. If the UNTIL clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes TRUE. The value is not tested before the first cycle of execution. On the second and all subsequent cycles of execution, the value is tested at the beginning of each cycle after the assignment of the loop variable. Use of the UNTIL version therefore always guarantees at least one cycle of execution.

7.6.5 The Iterative DO FOR Statement.

The iterative DO FOR statement is similar in intent and operation to the discrete DO FOR statement, except that the list of values that the loop variable may take on is replaced by an initial value, a final value, and an optional increment.

SYNTAX:

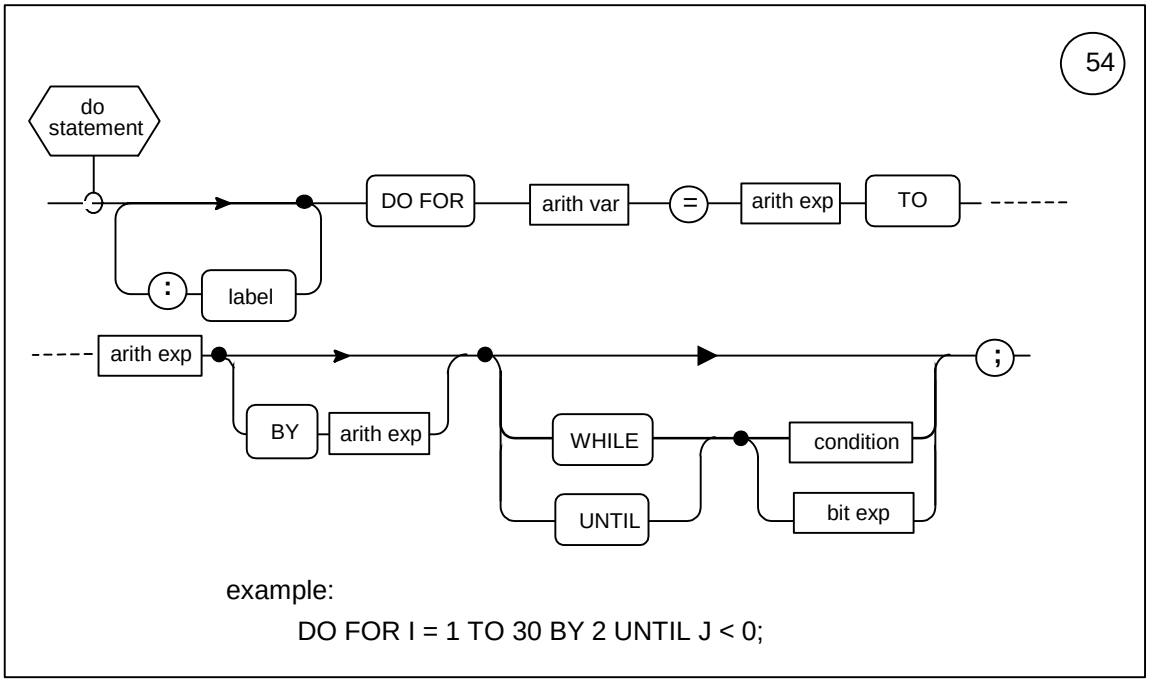


Figure 7-12 iterative DO FOR statement - #54

SEMANTIC RULES:

1. <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable. The initial loop variable, determined after all required subscripting and NAME dereferencing, is used throughout.
2. Each <arith exp> is any unarrayed integer or scalar expression. All are evaluated prior to the first cycle of execution of the group.

3. If a BY clause appears in the DO FOR statement, the value assigned to the loop variable prior to the K^{th} cycle of execution is equal to its value on the $K\text{-}1^{th}$ cycle plus the value of <arith exp> following the BY keyword (the “increment”).
4. Assignment of values to the loop variable follows the relevant assignment rules given in Section 7.3. In particular, if the loop variable is of integer type, and an initial value or increment is of scalar type, the latter will be rounded to the nearest integer in the assignment process. The effect of the loop variable assignment is identical to that of an ordinary assignment statement: the loop variable will retain the last value computed and assigned when the DO statement execution is completed.
5. After the value of the loop variable has been changed, it is checked against the value of the <arith exp> following the TO keyword (the “final value”).
6. If the sign of the increment is positive, the next cycle is permitted to proceed only if the current value of the loop variable is less than or equal to the final value.
7. If the sign of the increment is negative, the next cycle is permitted to proceed only if the current value of the loop variable is greater than or equal to the final value.
8. If the WHILE clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes FALSE. The value is tested at the beginning of each cycle of execution after the assignment of the loop variable. This implies that if <condition> or <bit exp> is FALSE prior to the first cycle of execution of the group, then the group will not be executed at all.
9. If the UNTIL clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes TRUE. The value is not tested before the first cycle of execution. On the second and all subsequent cycles of execution, the value is tested at the beginning of each cycle after the assignment of the loop variable. Use of the UNTIL version therefore always guarantees at least one cycle of execution.

7.6.6 The END Statement.

The END statement closes a DO…END statement group.

SYNTAX:

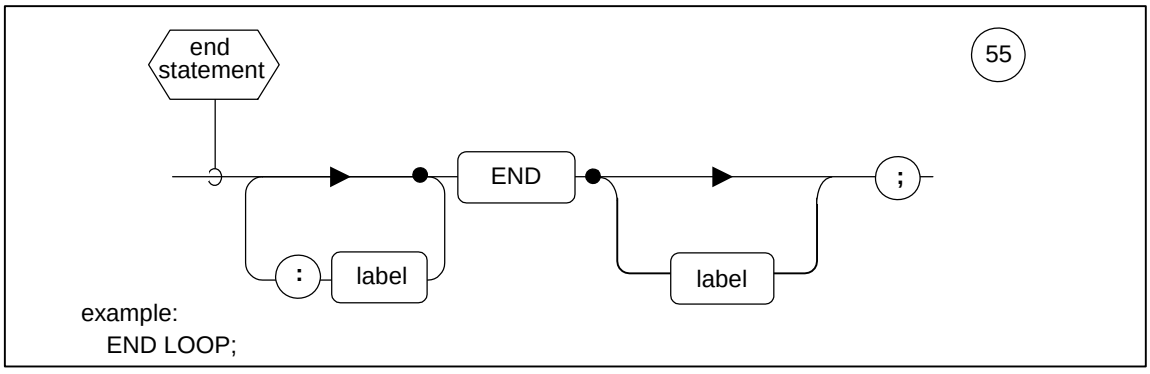


Figure 7-13 END statement - #55

SEMANTIC RULES:

1. If <label> follows the END keyword, then it must match a <label> on the <do statement> opening the DO…END group.
2. The <end statement> is considered to be part of the group, in that if it is branched to from a <statement> within the group, then depending on the form of the opening <do statement>, another cycle of execution of the group may begin. The END statement closing a DO CASE is not counted as another case.

7.7 Other Basic Statements.

Other <basic statement>s are the GO TO, “null”, EXIT, and REPEAT statements.

SYNTAX:

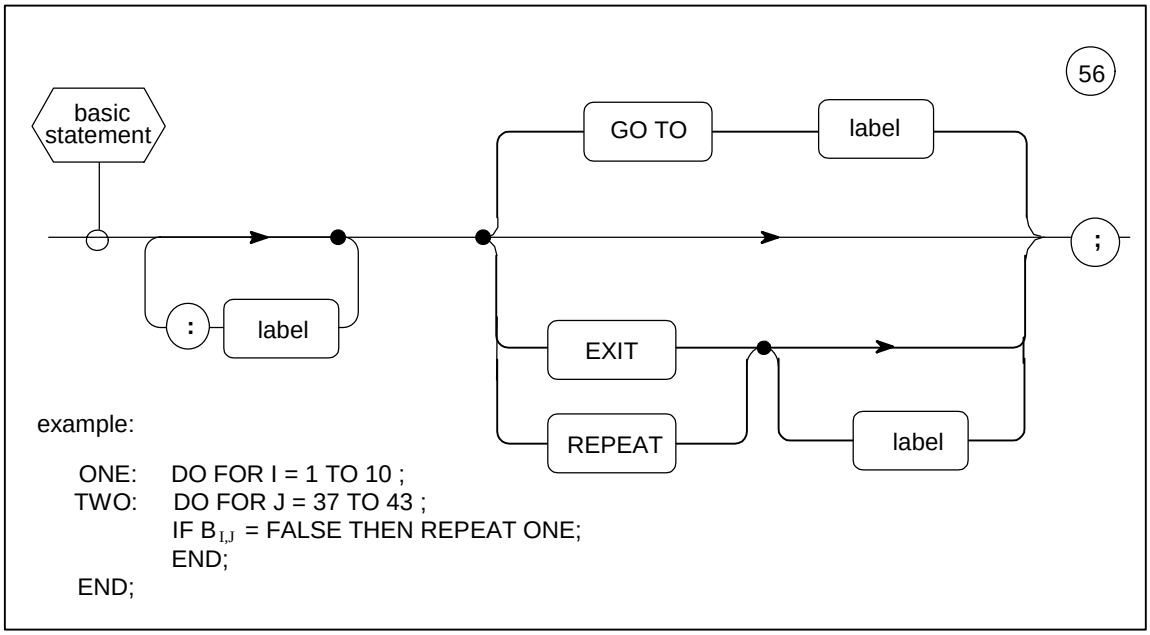


Figure 7-14 GO TO, “null”, EXIT, and REPEAT statements - #56

SEMANTIC RULES:

1. The GO TO <label> statement causes a branch in execution to an executable statement bearing the same <label>. The latter statement must be within same name scope as the GO TO statement. A GO TO statement may not be used to cause execution to branch into a DO…END group, or into or out of a code block.
2. The “null” statement (where no syntax except possible <label>s precede the terminating semicolon) has no effect at runtime.

3. The EXIT statement is only legal within a DO…END group, or within such groups nested. The form EXIT <label> controls execution relative to the enclosing DO…END group whose <DO statement> bears <label>. The form EXIT controls execution relative to the innermost enclosing DO…END group. Execution is caused to branch out of the DO…END group specified or implied, to the first executable statement after the group.
4. The REPEAT statement is only legal within a DO…END group opened with a DO FOR , DO WHILE, or DO UNTIL statement, or within such groups nested. The form REPEAT <label> controls execution relative to the enclosing such group whose <DO statement> bears <label>. The form REPEAT controls execution relative to the innermost such group. Execution is caused to abandon the current cycle of the DO…END group. If the condition of the opening <DO statement> are still satisfied, the next cycle of execution begins normally.
5. Code blocks (procedure, functions, etc.) may appear within DO…END groups. However, EXIT, REPEAT, and GO TO statements may not be used to cause execution to branch into or out of such code blocks.

This page is intentionally left blank.

8.0 REAL TIME CONTROL

HAL/S contains a comprehensive facility for creating a multi-processing job structure in a real time programming environment. At run time, a Real Time Executive (RTE) controls the execution of processes held in a process queue. HAL/S contains statements which schedule processes (enter them in the process queue), terminate them (remove them from the process queue), and otherwise direct the RTE in its controlling function. HAL/S also contains means whereby the use of data by more than one process at a time is managed in a safe, protected manner at specific, localized points within the data processes.

8.1 Real Time Processes and the RTE.

In HAL/S, a program or task may be scheduled as a process and placed in the process queue. Although the process created is given the same name as the program or task, it is important to distinguish the static PROGRAM or TASK block from the dynamic program or task process created. Two processes are actually involved in the creation of a process: the scheduling process, or "father"; and the scheduled process, or "son"[12].

A process is said to be either "dependent" or "independent", as designated when created. A program or task process is "dependent" if it is absolutely dependent for its existence upon the existence of its father. If a program process is "independent", its existence is independent of that of all other processes. If a task process is "independent", its existence is generally independent of that of all other processes with an important exception: the program process in whose static PROGRAM block the static TASK block of the task process is defined.

Each process in the RTE's process queue is at any instant in one of a number of states. For the purposes of this Section, the following states are defined:[13]

- "active" - a process is said to be in the active state if it is actually in execution. Depending on the implementation it may be possible for several processes to be in execution simultaneously.
- "wait" - a process is said to be in the wait state if it is ready for execution but the RTE has decided on a priority basis that its execution should be delayed or suspended.
- "ready" - a process is said to be in the ready state if it is in either the active or the wait states.
- "stall" - a process is said to be in the stall state if some as yet unsatisfied condition prevents it from being in the ready state.

The occurrence of a process being brought into the active state for the first time is called its "initiation".

12. except, of course, for the first or "primal" process which must be created by the RTE itself.
13. these states are not necessarily definitive of those actually existing in any particular implementation of the RTE.

Execution of a CLOSE or RETURN statement by an active process causes the following sequence of events:

1. CANCEL commands are issued for all DEPENDENT processes still on the process queue (see Section 8.4).
2. The process enters a stall state until all DEPENDENT processes have finished.
3. The current cycle is deemed finished. Control reverts to the RTE which may or may not remove the process from the process queue.

8.2 Timing Considerations.

In the HAL/S system, the RTE contains a clock measuring elapsed time ("RTE-clock" time). The time is measured in "machine units" (MUs) whose correspondence with physical time is implementation dependent. HAL/S contains several instances of timing expressions which in effect make reference to the RTE-clock.

Simultaneous occurrences produce implementation dependent results.

8.3 The SCHEDULE Statement.

The processes are scheduled (placed in the process queue) by means of the SCHEDULE statement. The statement has many variant forms and offers the following features:

- A process may be scheduled so that the RTE immediately places it in a ready state, or so that the RTE places it in a stall state pending some condition being satisfied.
- A process may be designated dependent or independent.
- The cyclic execution of a process may be specified.
- Conditions of future removal of a process from the process queue may be specified.

SYNTAX:

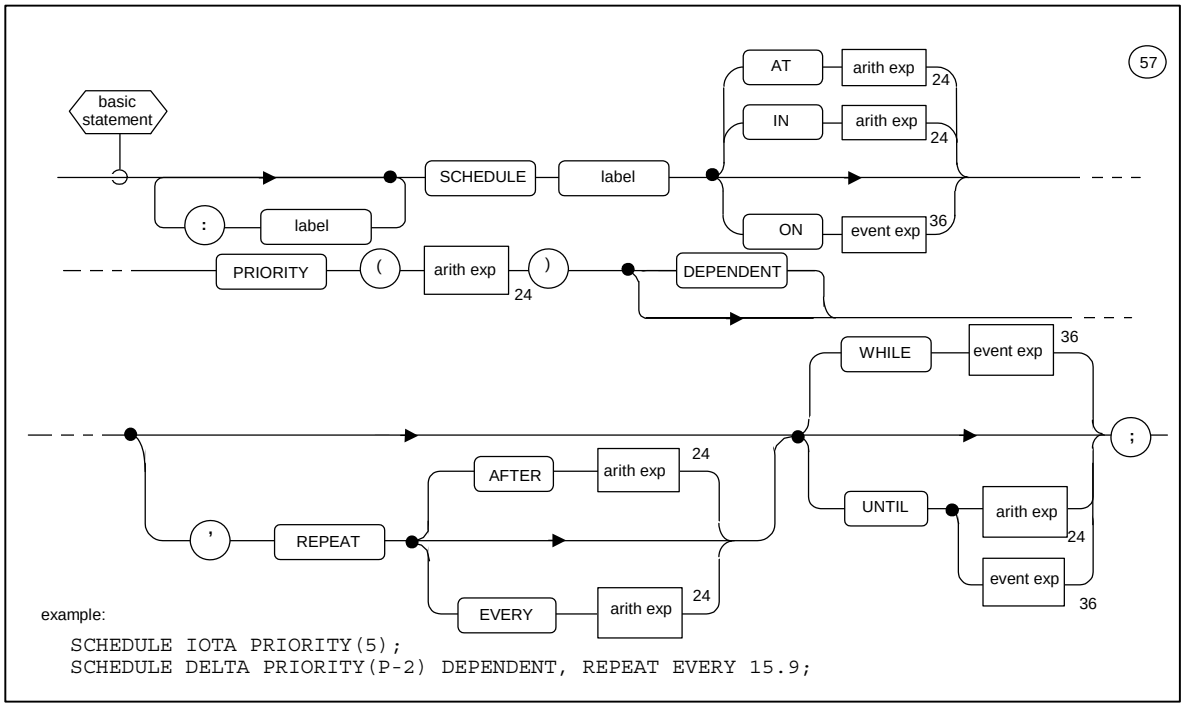


Figure 8-1 SCHEDULE statement - #57

SEMANTIC RULES:

1. SCHEDULE <label> schedules a program or task with the name <label>, placing a new process with the name <label> in the process queue. A run time error results if a process of that name already exists in the process queue. Unless otherwise specified, the RTE puts the new process in the ready state immediately after execution of the SCHEDULE statement.
2. The phrase IN <arith exp> is used to cause the process to be put in the stall state for a fixed RTE-clock duration. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then the process is put immediately in the ready state.
3. The phrase AT <arith exp> is used to cause the process to be put in the stall state until a fixed RTE-clock time. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than the current RTE-clock time and the REPEAT EVERY option is not specified, then the process is put immediately in the ready state. If the value is less than the current RTE-clock time and the REPEAT EVERY option was specified, then phased scheduling takes place. The process is put in a stall state until a future time computed by the expression CT + RE - ((CT -AT)MOD RE), where CT = current time, RE = REPEAT EVERY cycle time, and AT = originally specified AT time.

4. The phrase ON <event exp> is used to cause the process to be put in the stall state until some event condition is satisfied. Starting from the time of execution of the SCHEDULE statement, the <event exp> is evaluated at every "event change point"[14] until its value becomes TRUE. At that time the process is placed in the ready state. If the value of <event exp> is TRUE upon execution of the SCHEDULE statement, then the process is immediately put in the ready state.
5. The initiation priority is set by means of the phrase PRIORITY (<arith exp>) where <arith exp> is an unarrayed integer or scalar expression which is evaluated once on execution of the SCHEDULE statement. Scalar values are rounded to the nearest integral value. Its value must be consistent with the priority numbering scheme set up for any implementation, otherwise a run time error results. A priority value must be present in the SCHEDULE statement. Interpretation of priority is implementation dependent.
6. When the keyword DEPENDENT is specified, the process created by the SCHEDULE statement is dependent upon the continued existence of the scheduling process. Note, however, that a TASK process is always ultimately dependent upon the enclosing PROGRAM process. Thus, when scheduling a TASK from the PROGRAM level of nesting, the keyword DEPENDENT is redundant and need not be specified.
7. The REPEAT phrase of the SCHEDULE statement is used to specify a process which is to be executed cyclically by the RTE until some cancellation criterion is met. If the REPEAT phrase is not qualified, then the cycles of execution follow each other with no intervening time delay. To cause execution of consecutive cycles to be separated by a fixed intervening RTE-clock time delay, the qualifier AFTER <arith exp> is used. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero, then no time delay results. To cause the beginning of successive cycles of execution to be separated by a fixed RTE-clock time delay, the qualifier EVERY <arith exp> is used. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is such as to cause a cycle to try to start execution before the previous cycle has finished execution, then a run time error results.
8. Between the successive cycles of execution of a cyclic process, the process is put in a stall state and retains the machine resources the RTE reserved for it. It is not temporarily removed from the process queue.
9. The WHILE and UNTIL phrases provide a cancellation criterion for a cyclic process. Before the cyclic process is initiated, they also provide a means of removal of the process from the process queue. In this latter capacity, they also apply to noncyclic processes.

14. the meaning of an "event change point" is defined in Section 8.8.

10. The UNTIL <arith exp> phrase specifies a cancellation criterion based on RTE-clock time. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. For any process, cyclic or noncyclic, the following is true: if the value of <arith exp> is not greater than the current RTE-clock time, then the process is never entered in the process queue. Otherwise, a CANCEL command is issued if the RTE-clock equals <arith exp> while the process is still on the process queue (see Section 8.4).
11. The WHILE <event exp> phrase specifies a cancellation criterion based on an event condition. For any process, cyclic or non-cyclic, the following is true: if the value of <event exp> is FALSE at the time of execution of the SCHEDULE statement, then the process is never placed in the process queue. If not, then <event exp> is evaluated at every “event change point” until its value becomes FALSE. At this time a CANCEL command is issued if the process is still on the process queue (see Section 8.4).
12. The UNTIL <event exp> phrase also specifies a cancellation criterion based on an event condition. However, it differs fundamentally from the WHILE <event exp> phrase in that it always allows at least one cycle of a cyclic process to be executed. Consistent with this, the phrase has no meaning and, therefore, no effect in the case of a non-cyclic process. For a cyclic process, the value of the <event exp> is evaluated at every “event change point” from the time of execution of the SCHEDULE statement.

    If <event exp> becomes TRUE prior to the end of the first cycle, a CANCEL command is issued at the end of the first cycle. Otherwise, if <event exp> becomes TRUE while the process is still on the process queue, a CANCEL command is issued at that time (see Section 8.4).

8.4 The CANCEL Statement.

Cancellation of a process may be the result of the enforcement of a cancellation criterion in the SCHEDULE statement which created the process, or, alternatively, may be the result of executing a CANCEL statement.

SYNTAX:

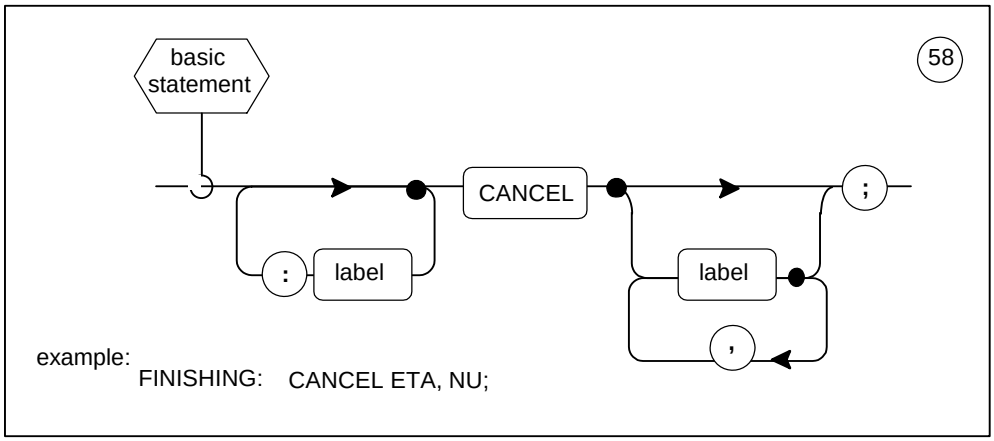


Figure 8-2 CANCEL statement - #58

SEMANTIC RULES:

1. CANCEL <label> causes the cancellation of the process <label>. A run time error results if the process queue contains no process with that name.[15] The CANCEL statement can be used to cancel any number of processes simultaneously.
2. If the CANCEL statement has no <label>, cancellation of the process executing the CANCEL statement is implied.
3. If at the time of execution of the CANCEL statement, a process to be canceled has not yet been initiated, then the process is merely removed from the process queue. This applies to both cyclic and non-cyclic processes.
4. If at the time of execution of the CANCEL statement, a process to be canceled has already been initiated, then the following ensues: if the process is non-cyclic and it has already been initiated, the CANCEL statement has no effect; if the process is cyclic, then the process is removed from the process queue at the end of the current cycle of execution.

8.5 The TERMINATE Statement.

The termination of a process results in the immediate[16] cessation of execution of the process, TERMINATEs of dependents, and removal from the process queue. The TERMINATE statement is used to direct the RTE to terminate specified processes or the process issuing the TERMINATE.

SYNTAX:

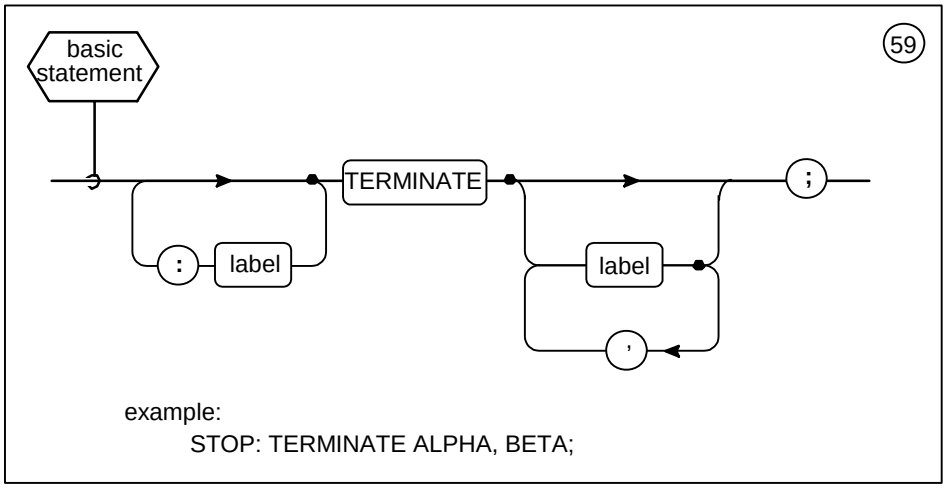


Figure 8-3 TERMINATE statement - #59

15. the default action taken by the Error Recovery Executive for this and other similar errors may be to ignore the error.
16. subject, of course, to implementation dependent safety restraints.

SEMANTIC RULES:

1. TERMINATE <label> causes termination of the process <label>. A run time error results if a process of that name is not in the process queue, or if it is not a dependent son of the process currently executing the TERMINATE statement. The TERMINATE statement can be used to terminate any number of processes simultaneously.
2. If the TERMINATE statement has no <label>, termination of the process currently executing the TERMINATE statement is implied.

8.6 The WAIT statement.

The WAIT statement allows the user to cause the RTE to place a process in the stall state until some condition is satisfied.

SYNTAX:

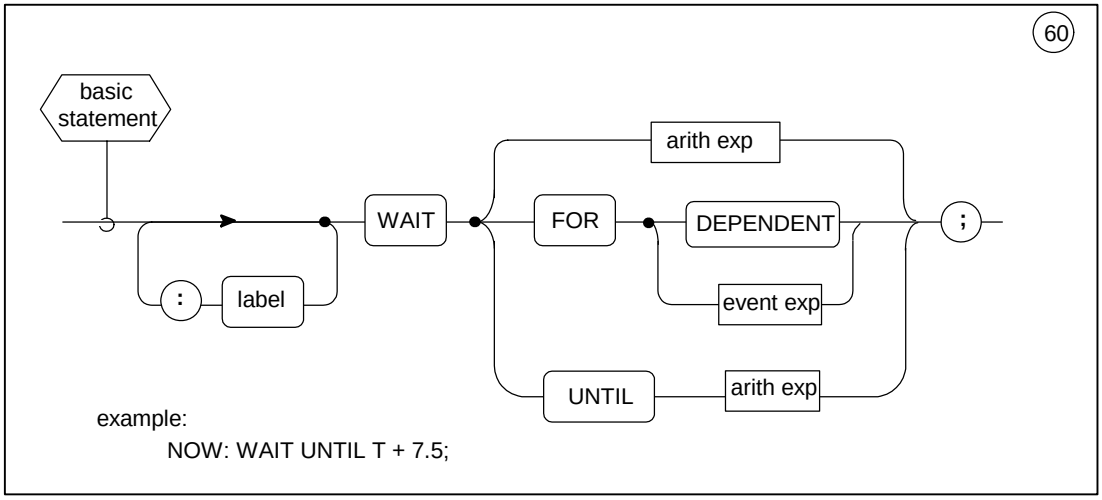


Figure 8-4 WAIT statement - #60

SEMANTIC RULES:

1. The WAIT <arith exp> version specifies that the process executing the WAIT statement is to be placed in the stall state for an RTE-clock duration fixed by the value of the expression. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the WAIT statement. If the value is not greater than zero, the WAIT statement has no effect.
2. The WAIT UNTIL <arith exp> version specifies that the process executing the WAIT statement is to be placed in the stall state until an RTE-clock time fixed by the value of the expression. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the WAIT statement. If the value is not greater than the current RTE-clock time, the WAIT statement has no effect.
3. The WAIT FOR DEPENDENT version specifies that the process executing the WAIT statement is to be placed in the stall state until all its dependent sons have terminated. If there are no such processes, the WAIT statement has no effect.

4. The WAIT FOR <event exp> version specifies that the process executing the WAIT statement is to be placed in the stall state until an event condition is satisfied. Starting from the time of execution of the WAIT statement, the <event exp> is evaluated at every “event change point” until its value becomes TRUE, whereupon the process is returned to the READY state. If the value of <event exp> is TRUE upon execution of the WAIT statement, then the statement has no effect.

8.7 The UPDATE PRIORITY Statement.

The SCHEDULE statement which creates a process can also specify the priority of its initiation. At any time between the scheduling and the termination of the process, that priority may be changed by means of the UPDATE PRIORITY statement.

SYNTAX:

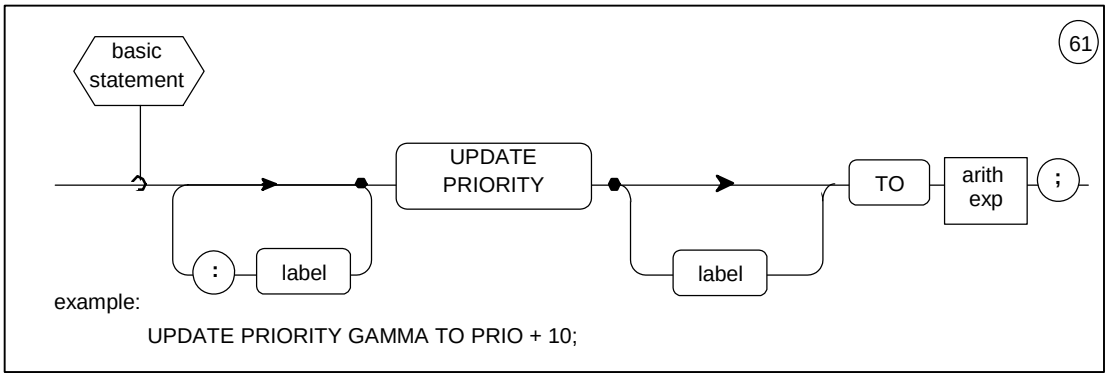


Figure 8-5 UPDATE PRIORITY statement - #61

SEMANTIC RULES:

1. UPDATE PRIORITY <label> is used to change the priority of the process with the name <label>. The new priority is given by the value of <arith exp>. <arith exp> is an unarrayed integer or scalar expression whose value must be consistent with the priority numbering scheme set up for any implementation, otherwise, a run time error results. Scalar values are rounded to the nearest integral value. A run time error results if there is no process with the name <label> in the process queue.
2. UPDATE PRIORITY with no <label> specification is used to change the priority of the process executing the UPDATE PRIORITY statement. <arith exp> has the same meaning as before.

8.8 Event Control.

Although a formal specification of events and event expressions has already been discussed in Sections 4 and 6.3, the Specification has not yet made their purpose clear in the context of real time programming. Superficially, event variables are closely akin to boolean variables in that they are binary valued. Conceptually, the two forms of HAL/S events (latched and unlatched) may be thought of as the software counterparts of hardware discrete and timing lines, respectively.

- a latched event may be thought of as a boolean system state which may be SET or RESET by appropriate actions, or momentarily changed for signaling purposes.

- an unlatched event event may be thought of as the software counterpart of a timing line which is used purely for signaling - it is normally FALSE but becomes TRUE momentarily when a signal action is executed.

This analogy is no accident, since event variables can actually form the interface between HAL/S software and such hardware control signals. The design and operation of this interface is implementation dependent.

At any instant of time the RTE may be viewed as having a knowledge of all existing events. Whenever the value of an event changes, the RTE senses this so-called “event change point”, and may in response perform the evaluation of certain <event exp>s. Depending on the results of the evaluations, the states of one or more processes may be changed. This response of the RTE to changes in event variables is termed an “event action”. The value of an event variable can change in response to the environment external to the HAL/S software; depending upon the type of event (see SEMANTIC RULES), a SET, a RESET, or a SIGNAL statement may also be used to alter the state of an event variable. The only event change actions possible are to ready or cancel one or more processes.

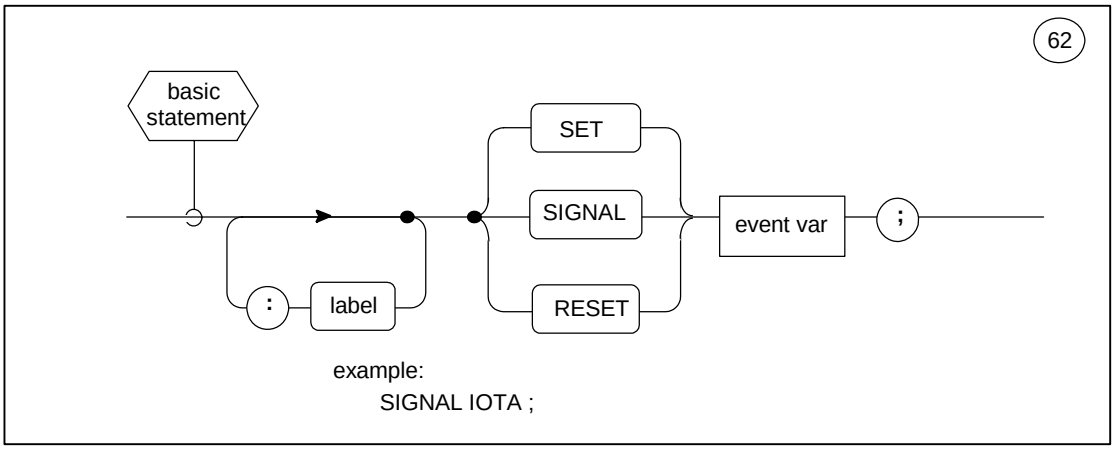


Figure 8-6 SET, SIGNAL, and RESET statements - #62

GENERAL SEMANTIC RULE:

1. <event var> denotes any unarrayed event variable, subscripted or unsubscripted.

SEMANTIC RULES (latched <event var>s):

1. SET changes the value of the <event var> to TRUE and initiates all event actions depending upon the TRUE state of this event. No action is taken if the <event var> is already TRUE.
2. RESET changes the value of the <event var> to FALSE and initiates all event actions depending upon the FALSE state of this event. No action is taken if the <event var> is already FALSE.
3. SIGNAL does not change the state of a latched event.
4. If a latched event is TRUE, SIGNAL initiates all event actions depending upon the FALSE state of this event.

5. If a latched event is FALSE, SIGNAL initiates all event actions, depending upon the TRUE state of this event.

SEMANTIC RULES (unlatched <event var>s):

1. SET and RESET are illegal for unlatched <event var>s.
2. When used in a <bit expression>, an unlatched event variable is equivalent to a literal “FALSE”.
3. SIGNAL initiates all event actions depending upon the TRUE state of this event. Note that when an event expression depends upon a logical product of multiple <event var>s, at most one such <event var> can be unlatched if the event action is ever to be taken.

SUMMARY:

		SET	RESET	SIGNAL
unlatched event		illegal	illegal	Take all event actions depending on TRUE state of <event var>
latched event	old value is FALSE	1. Set event state to TRUE 2. Take all event actions depending on TRUE state of <event var>	no action	Take all event actions depending on TRUE state of <event var>
latched event	old value is TRUE	no action	1. Set event state to FALSE 2. Take all event actions depending on FALSE state of <event var>	Take all event actions depending on FALSE state of <event var>

Table 8-1 Latched and Unlatched Events in Set, Reset, and Signal Statements

8.9 Process-events.

Every program or task block has associated with it a “process-event” of the same name. This process-event behaves in every way like a latched event except that it may not appear in SET, RESET, or SIGNAL statements. Its purpose is to indicate the existence of its associated program or task process. If a process of the same name as the process-event exists in the process queue, the value of the process-event is TRUE, otherwise, it is FALSE.

8.10 Data Sharing and the UPDATE Block.

The UPDATE block provides a controlled environment for the use of data variables which are shared by two or more processes. If controlled sharing of certain variables is desired, they must possess the LOCK (N) attribute, where N indicates the “lock group” of the variable (see Section 4.5). LOCKed variables may only be used inside UPDATE blocks. A LOCKed variable appearing inside an UPDATE block is said to be “changed” within the block if it appears in one or more statements which may change its value (the left-hand side of an assignment, for example). It is said to be “accessed” if it only appears in contexts other than the above.

A formal specification of the UPDATE block appears in Section 3.4. The manner of operation of an UPDATE block is implementation dependent, but is such as to provide certain safety measures.

OPERATIONAL RULES:

1. If two processes both require variables from the same lock group to be changed, then the first process entering its UPDATE block must complete execution of the block before the other process can enter its own UPDATE block. The second process is placed in a stall state for the duration.
2. If one process entering an UPDATE block requires a variable(s) with the attribute LOCK(*) to be changed, then the situation is equivalent to one in which the process requires use of a variable from every lock group.
3. If only one of the processes requires a variable of a lock group to be changed, the other merely requiring it to be accessed, then depending on the implementation, either Rule 1 or 2 holds, or some overlap in execution of the two processes’ UPDATE blocks is allowed. The nature of such overlap must be such as to provide exclusive use of the lock group by the process requiring its change between the point where the variable is changed and the close of the UPDATE block.
4. If both processes only require a variable of the same lock group accessed, then execution of the two processes’ UPDATE block may be allowed to overlap depending upon implementation.
5. If there are several simultaneous conflicts in using shared variables because of the participation of more than two processes, or more than one lock group, then the most restrictive of Rules 1 through 4 required is applied to resolve the conflicts.

This page is intentionally left blank.

9.0 ERROR RECOVERY AND CONTROL

References to so-called ‘run time errors’ have been made elsewhere in this Specification. Such errors arise at execution time through the occurrence of abnormal hardware or system software conditions. Each HAL/S implementation possesses a unique collection of such errors. The errors in the collection are said to be “system-defined”. In any implementation, every possible system-defined error is assigned a unique “error code”. In addition, a number of other legal error codes not assigned to system-defined errors may exist. These can be used by the HAL/S programmer to create “user-defined” errors. All run time errors, both system- and user-defined, are classified into “error groups”. The error code for an error consists of two positive integer numbers; the first representing the error group to which it belongs, and the second uniquely identifying it within its group. The method of classification is implementation dependent.

At run time an Error Recovery Executive (ERE) senses errors, both system-defined and user-defined, and determines what course of action to take. For every error group, a standard system recovery action is defined which the ERE will take unless error recovery has been otherwise directed by the user. Depending upon the error and the implementation, the standard system recovery action may be to terminate execution abnormally, to execute a fix-up routine and continue, or to ignore the error.

In a real time programming context, every process in the process queue has a separate, independent “error environment” which is continuous from the time of initiation of the process to the time of its termination. At any instant of time the “error environment” of a process is the totality of error recovery actions in force at that time for all possible errors. At the time of initiation of the process, the standard system recovery action is in force for all errors.

HAL/S possesses two error recovery and control statements. The ON ERROR statement is used to modify the error environment of a process at any time during its life. The SEND ERROR statement is used for the two-fold purpose of creating user defined error occurrences, and simulating system-defined error occurrences.

9.1 The ON ERROR Statement.

The error environment upon entry into a code block (other than PROGRAM or TASK) is unchanged from that of the previous statement executed. If a code block changed the error environment, the error environment upon entry into the code block is restored upon exit from the code block.

The ON ERROR statement is used to change the error environment prevailing at the time of its execution. It can change the error recovery action for one selected error code, for one selected error group, or for all groups simultaneously. There are two basic forms of the statement: ON ERROR and OFF ERROR.

If an ON ERROR with a given specification is executed in a particular code block, then the modified recovery action remains in force until one of three things happens:

- the modification is superseded by execution of a second ON ERROR with the same error specification.
- the modification is removed by execution of an OFF ERROR with the same error specification, the recovery action thereupon reverting to that in force on entry into the code block.
- the modification is automatically removed by exit from the code block.

SYNTAX:

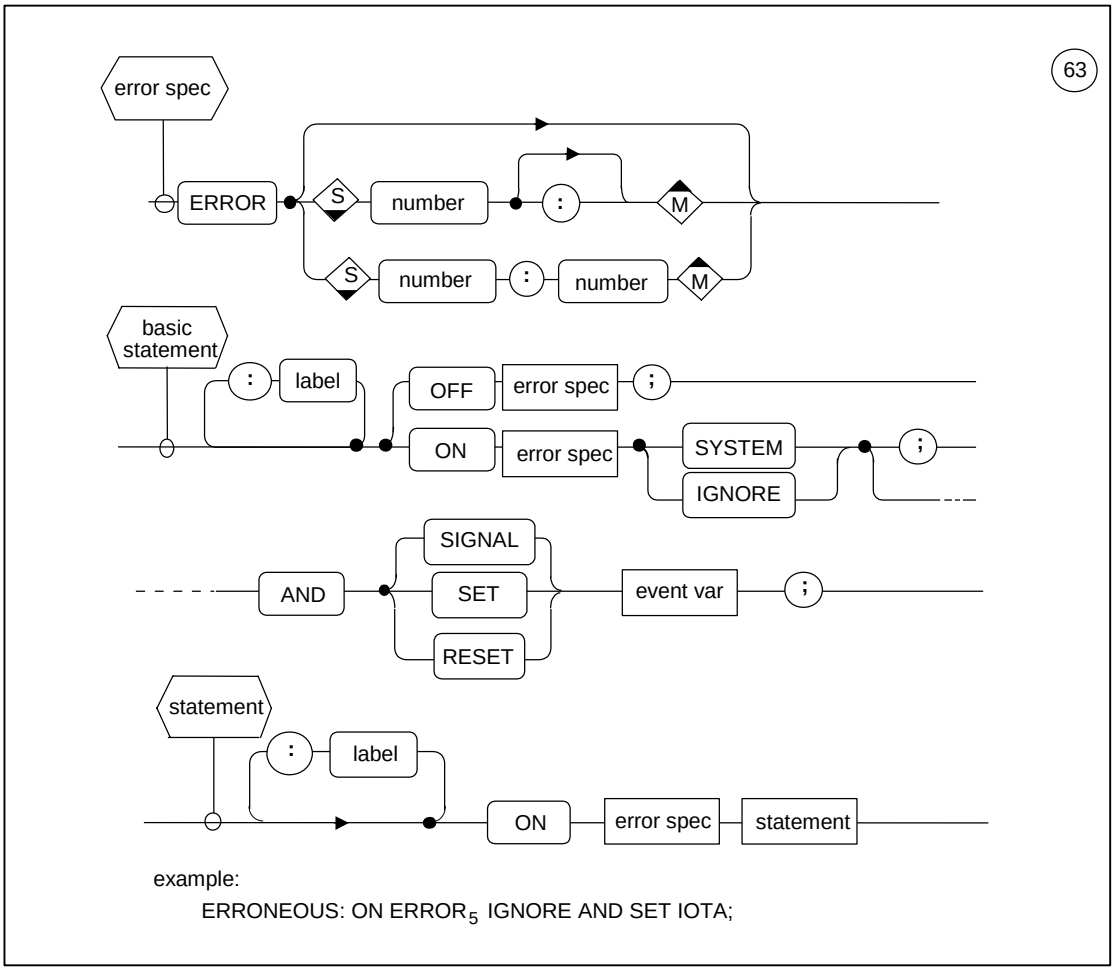


Figure 9-1 ON ERROR statement - #63

SEMANTIC RULES:

1. The ON ERROR statement consists of two parts: a specification of an error action to be taken by the ERE, preceded by an <error spec> specifying the error number and the error group or groups to which the action is to apply.
2. There are three forms of <error spec> for specifying either all error groups, a selected error group, or a selected error code:
   - The form of <error spec> without subscript is used to specify all error groups.
   - The subscript construct <number> with optional following colon is used to specify a selected <error group>. The value of <number> is restricted to the set of error group numbers defined for a particular implementation.
   - The subscript construct <number>: <number> is used to specify a selected error code. The leftmost <number> designates the error group number; the rightmost <number> the selected error number within the group. Values are restricted to the set of error codes defined for a particular implementation.

3. The form ON ERROR…. specifies the modification of the error recovery actions for the given <error spec>. OFF ERROR…. specifies the removal of a modification previously activated in the name scope for the same <error spec>. If no such modification exists, the OFF ERROR is, effectively, a no-operation.
4. The presence of the IGNORE clause specifies that in the event of occurrence of a specified error, the ERE is to take no action other than allow execution to proceed as if the error had not occurred. The IGNORE action may not be permitted for certain errors.
5. The presence of the SYSTEM clause specifies that in the event of the occurrence of a specified error, the ERE is to take the standard system recovery action.
6. The form ON ERROR…<statement> specifies that <statement> is to be executed on the occurrence of a specified error <statement> may optionally be labeled. However, such labels may only be referenced by EXIT or REPEAT statements within the (compound) <statement> thus labeled. After execution of <statement>, execution normally restarts from the executable statement following the ON ERROR statement. Execution of <statement> itself may of course modify this.
7. It is important to note that the form ON ERROR…<statement> is itself a <statement> while other forms of ON ERROR are <basic statement>s. The form ON ERROR…<statement> may therefore not be the true part of an IF…THEN…ELSE statement.
8. If an ON ERROR possesses a SYSTEM or IGNORE clause, it may also possess an additional SIGNAL, SET, or RESET clause. The purpose is to cause the value of an <event var> to be changed on the occurrence of a specified error. Its semantic rules are the same as those described for the corresponding SIGNAL, SET, or RESET statements in Section 8.8. Note that if <event var> contains a subscript expression, then that expression will be evaluated at the time of execution of the ON ERROR statement, not on the occurrence of the error.

PRECEDENCE RULE:

1. An ON ERROR executed within a code block always totally supersedes an ON ERROR executed before entering the code block.
2. Within a code block the action specified by an ON ERROR is only superseded by another if the two <error spec>s are of identical form. Similarly, an OFF ERROR nullifies the effect of a previous ON ERROR only if the two <error spec>s are of identical form. However, <u>different</u> forms of <error spec> may involve the same error group or error code. It is logically possible for up to three ON ERRORs, each with a different form of <error spec> as described in Rule 2 above, to be active simultaneously and involve the <u>same error code</u>. The ON ERROR precedence order for determining the recovery action in the event of an error occurrence is as follows:

Error Specification	<error spec> subscript construct	Precedence
		LAST
all groups	–	1
selected group	<number>:or <number>	2
selected error code	<number>:<number>	3
		FIRST

Table 9-1 Precedence Rules for ON ERROR

9.2 The SEND ERROR Statement.

The SEND ERROR statement is used to announce a selected error condition to the ERE. If the error selected is 'system-defined', then in effect that error is being simulated.

SYNTAX:

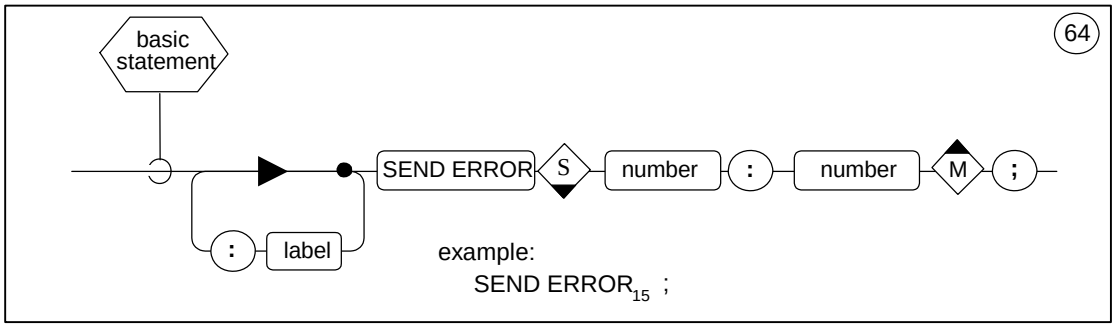


Figure 9-2 SEND ERROR statement - #64

SEMANTIC RULES:

1. <number> : <number> is a subscript construct consisting of two unsigned integer literals. The leftmost <number> designates the error group to which the selected error condition belongs. The rightmost <number> denotes the error number within the designated group. Values are restricted to the set of error codes defined for a particular implementation. If the error code corresponds to a system-defined error, then that error is simulated by the ERE. Simulation of certain system-defined errors may not be permitted.
2. The action taken by the ERE after announcement of the selected error condition is dictated by the error environment prevailing at the time of execution of the SEND ERROR statement.

10.0 INPUT/OUTPUT STATEMENTS

The HAL/S language provides for two forms of I/O: sequential I/O with conversion to and from an external character string representation, and random-access record-oriented I/O.

All HAL/S I/O is directed to one of a number of input/output “channels”. These channels are the means to interface HAL/S software with external devices in a runtime environment. In any implementation, each channel is assigned a unique unsigned integer identification number.

The input/output statements described in this Section are intentionally general-purpose. They provide a basic support facility for applications programming on the Shuttle project. Specialized hardware-oriented I/O commands may be created via features of the HAL/S Systems Language.

10.1 Sequential I/O Statement.

All sequential I/O in HAL/S is to or from character-oriented files. HAL/S pictures these files as consisting of lines of character data similar to a series of printed lines or punched cards. An “unpaged” file simply consists of an unbroken series of such lines. In a “paged” file the lines are blocked into pages, each a fixed, implementation dependent number of lines in length. The choice of paged or unpaged file organization for each sequential I/O channel is specified in an implementation dependent manner.

HAL/S pictures the physical device as moving across the file a read or write “device mechanism” which actually performs the data transfer. The device mechanism has at every instant a definite column and line position on the file. The act of transmitting one character to or from the file is followed by the positioning of the device mechanism to the next column on the same line. When the end of the line is reached the device mechanism moves on to the first (leftmost) column of the next line.

The HAL/S sequential I/O statements are the READ, READALL, and WRITE statements. Within these statements I/O control functions can be used to cause explicit positioning of the device mechanism on the file.

10.1.1 The READ and READALL Statements.

The sequential input of data is accomplished in HAL/S by employing either a READ or READALL statement. The choice depends upon the format of the character input and the conversions (if any) which are to be performed.

A READALL statement is used whenever arbitrary character string images are to be input without conversion; otherwise, READ is used.

<format list>s may be used with READ statements when data is not in a standard external format; e.g., if two numbers are located in consecutive columns without separation.

SYNTAX:

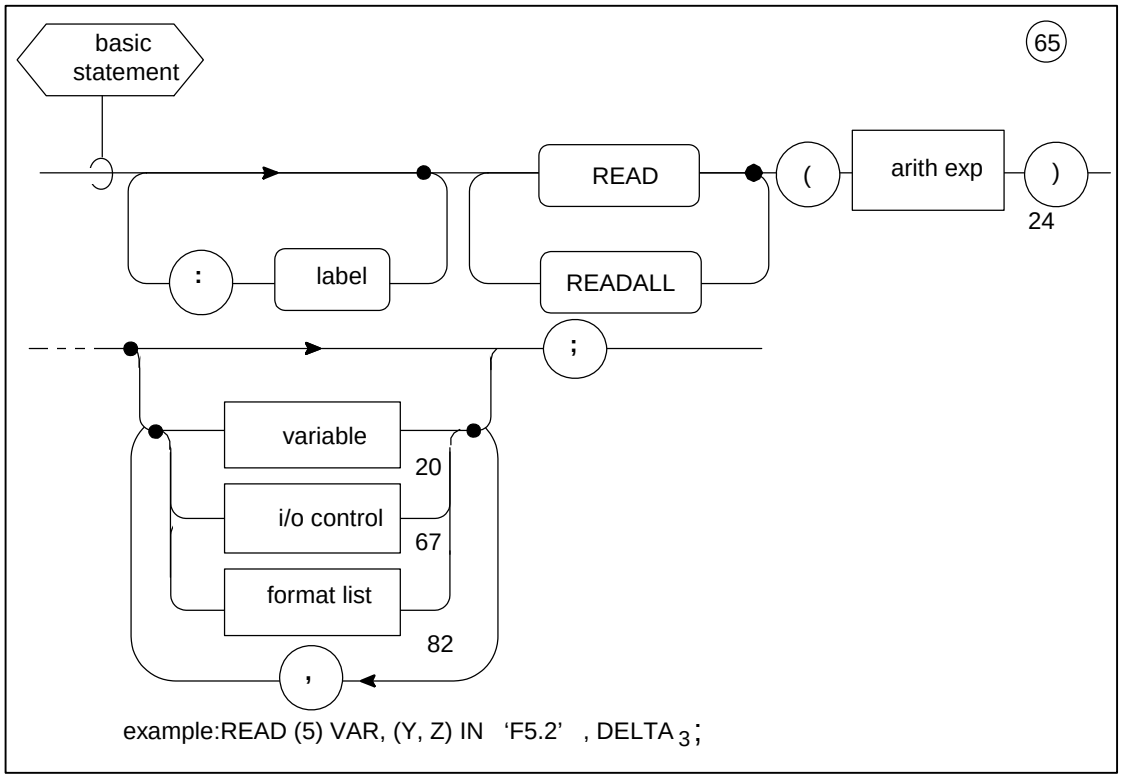


Figure 10-1 READ and READALL statements - #65

GENERAL SEMANTIC RULES:

1. <arith exp> is an unarrayed scalar or integer arithmetic expression. The value is treated as an integer: scalar values are rounded to the nearest integer prior to use. The value must represent a legal I/O channel number.
2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.
3. Unless overridden by explicit <i/o control> or <format list>, the device mechanism is automatically moved to the leftmost column position and advanced to the next line prior to reading the first <variable>. A SKIP, LINE, or PAGE before the first <variable> overrides the automatic line advancement. A TAB or COLUMN overrides the automatic column position.
4. An unexpected end of file reached during the reading of data from the input file causes a runtime error.
5. <variable>s are read in order. Each <variable>'s subscript is evaluated just prior to its input.

SEMANTIC RULES (READALL version):

1. <variable> may be any character or structure variable in an assignment context. This specifically excludes input parameters of functions and procedures. If it is of structure type, all the terminals of the template it references must be of character type. In this case, no nested structure template references are allowed.
2. If <variable> is an array or structure each element thereof is filled sequentially in its "natural sequence".
3. Data is read from the input file character by character from left to right, each <variable> element being filled in turn. Filling of an element is completed either when the end of a line on the file is reached, or when the element has reached its declared maximum length, whichever happens sooner.
4. <format list> may not be used with READALL.

SEMANTIC RULES (READ version):

1. <variable> is any variable which may be used in an assignment context. This specifically excludes input parameters of functions and procedures.
2. If <variable> is a vector or matrix, or an array or structure, each element thereof is filled sequentially in its "natural sequence".
3. When reading data specified in a <format list> the device mechanism is positioned by the <format list>. All the characters in the field determined by the format are transmitted and converted to the internal HAL/S data type. If the width of the specified field is greater than the number of characters remaining on the line, an implementation dependent mechanism is invoked.
4. In the absence of a <format list>, the device mechanism (subject to <i/o control>) scans the input file left to right, from line to line, looking for fields of contiguous characters separated by commas, semicolons, or blanks. Each field found is in turn transmitted and converted from its standard external format to an appropriate HAL/S data value. Fields may not cross line boundaries except when reading character strings.
5. When not under control of a <format list>, a semicolon field separator encountered during a normal sequential scan to fill a variable element terminates the READ statement as follows:
   - The current <variable> element is left unchanged;
   - All remaining <variable>s in the statement are unchanged;
   - All remaining control functions in the statement are ignored.

   <i/o control> functions can force the device mechanism over the semicolon without causing early termination.
6. When not under the control of a <format list>, a null field is transmitted whenever a comma or semicolon is detected when data is expected. This occurs when a comma or semicolon is:
   - preceded by a comma or semicolon;
   - preceded by one or more blanks following the last comma or semicolon.

When under control of a <format list>, a null field is transmitted and an error sent whenever the field being read is entirely blank.

A null field causes the corresponding variable element to remain unchanged following transmission.

7. For READ statements, fields must either be read using <format list> or else they must appear in a standard external format. A list of standard external formats is given in Appendix E. A type mismatch causes a runtime error.

10.1.2 The WRITE Statement.

The sequential output of data is accomplished in HAL/S by employing the WRITE statement.

SYNTAX:

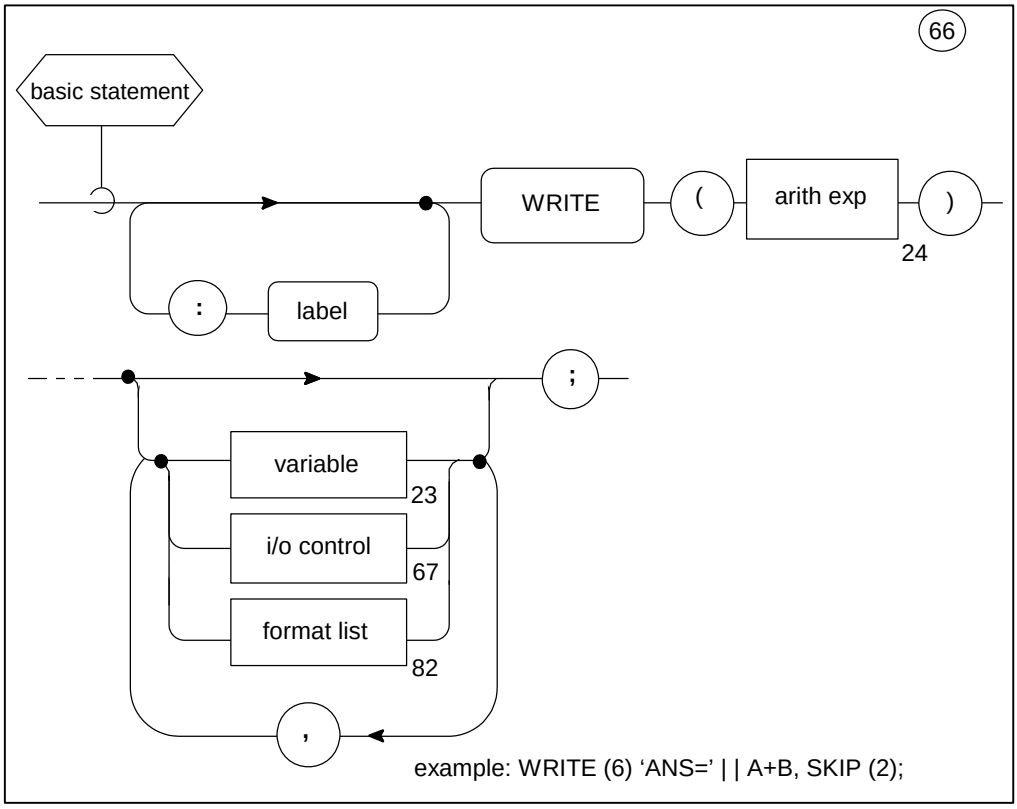


Figure 10-2 WRITE statement - #66

SEMANTIC RULES

1. <arith exp> is an unarrayed integer or scalar arithmetic expression. The value is treated as an integer: scalar values are rounded to the nearest integer prior to use. The value must represent a legal I/O channel number.
2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.
3. There are no semantic restrictions on <expression>.

4. If <expression> is of vector or matrix type, or is an array or structure, then each element thereof is transmitted sequentially in its “natural sequence”.
5. Unless overridden by explicit <i/o control>, or by a <format list>, the device mechanism is automatically moved to the leftmost column position and advanced to the next line prior to transmitting the first <expression>. A SKIP, LINE, or PAGE <i/o control> before the first <expression> overrides the automatic line advancement. A TAB or COLUMN <i/o control> overrides the automatic column positioning.
6. Each <expression> in turn is converted to its standard external format before being transmitted to the output file. A list of standard external formats is given in Appendix E.
7. <format list> may be used to output data in a non-standard form.
8. <format list> may specify additional <expression>s to be transmitted in non-standard formats.

   Example:

   Output INTEGERS K1 and K2 + K3 in columns 1-5 and 6-10, respectively:

```
WRITE(6) (K1, K2+K3) IN '2I5';
```

9. When not under control of a <format list>, the device mechanism is moved to the right by an implementation dependent number of columns between the transmissions of two consecutive elements. If a TAB or COLUMN <i/o control> separates two consecutive <expression>s then this overrides the automatic movement between transmission of the last element of the first <expression> and the first element of the second <expression>.
10. When a line has been filled to the point where the next converted output field will not fit in the remaining columns, a wrap-around condition occurs. The actions taken in such a case are implementation dependent.

10.1.3 I/O Control Functions.

An I/O control function is introduced into a READ, READALL, or WRITE statement to cause explicit movement of the device mechanism. Note that the interpretation of each I/O control function differs depending upon whether the file is paged or unpaged.

SYNTAX:

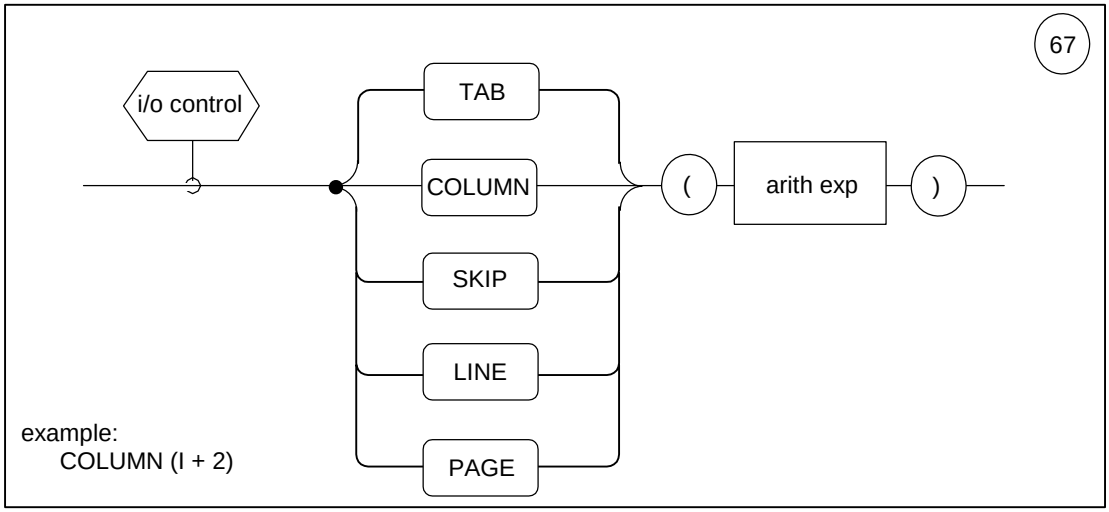


Figure 10-3 i/o control function - #67

SEMANTIC RULES:

1. <arith exp> is an unarrayed scalar or integer arithmetic expression specifying a value to the control function. The value is treated as an integer: scalar values are rounded to the nearest integer prior to use. In the following rules, let the value of <arith exp> be denoted by K.
2. TAB(K) specifies relative movement of the device mechanism across the current line by K character positions (columns). The motion is to the right (increasing column index) if K is positive, to the left if K is negative. Positioning to negative or zero column index values, or to a positive index greater than an implementation dependent maximum causes a runtime error.
3. COLUMN(K) specifies absolute movement of the device mechanism to column K of the current line. Values of K may range from 1 to an implementation dependent maximum value. Column indices outside the legitimate range cause a runtime error.
4. SKIP(K) specifies line movement relative to the current line of the file. A positive value of K will cause forward movement. Subject to implementation and hardware restrictions, backward movement is indicated by a negative value of K. Error conditions will be indicated if a skip causes movement past either end of the file, or movement in violation of any implementation restriction on the direction of the skip.
5. LINE(K) specifies line movement to a specified line number, K. Two interpretations occur depending upon whether the file is paged or unpaged.
   - Paged files - LINE(K) advances the file unconditionally. K may not be less than 1 or greater than the implementation and hardware dependent number of lines per page; otherwise, an error condition will be indicated. If K is not less than the current line number, the new print position is on the current page; if K is less than the current line number, the device mechanism is advanced to line K of the next page.

- Unpaged files - LINE(K) positions the device mechanism at some absolute line number in the file. On input K must be greater than zero, but not greater than the total number of lines in the file. On output, K must merely be greater than zero. In either case, values outside the indicated ranges cause runtime errors. Depending upon the implementation, values of K causing backwards movement may be illegal.

6. PAGE(K) is only applicable to paged files and specifies page movement relative to the current page. If K is positive the movement is forward, toward the end of the file. Depending upon the implementation, negative page values may or may not be legal. The line value relative to the beginning of the page remains unchanged.

10.1.4 FORMAT Lists.

FORMAT lists present a powerful way to perform I/O operations with complete explicit control of all conversion and layout functions.

SYNTAX:

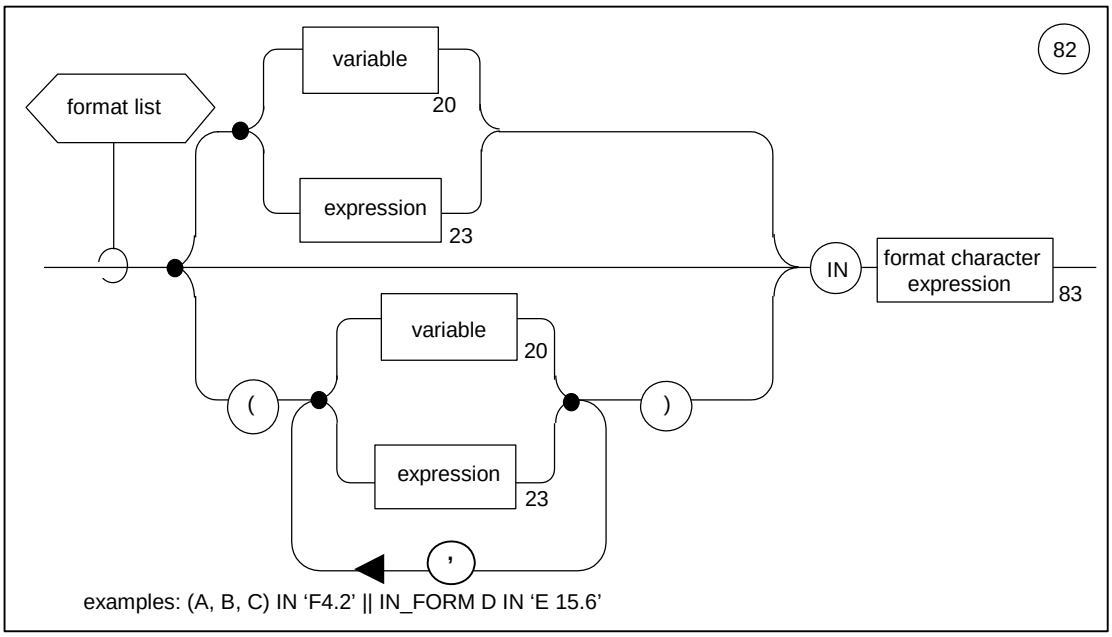


Figure 10-4 FORMAT lists - #82

SEMANTIC RULES:

1. A <format list> used with a READ may only contain <variable>s, not <expression>s.
2. <format character expression> is any character expression. A runtime check is made for legality.
3. All variables in the <format character expression> are evaluated before any I/O takes place involving the FORMAT list.

4. Each <expression> or <variable> is handled according to <format character expression>. If the <expression> or <variable> represents an aggregate of elements, then each element is handled sequentially in its natural sequence.

   Example:

```
DECLARE V VECTOR INITIAL(2.12, 3.4, -7)
F CHARACTER(16) INITIAL('F4.2, F5.1, F4.1');
```

   then:

```
 WRITE(6) V IN F
                2.12bb3.4-7.0
produces:       ↑    ↑   ↑
      column    1    5   10
```

10.1.4.1 FORMAT Character Expressions.

FORMAT character expressions determine how items in FORMAT lists are read or written.

SYNTAX:

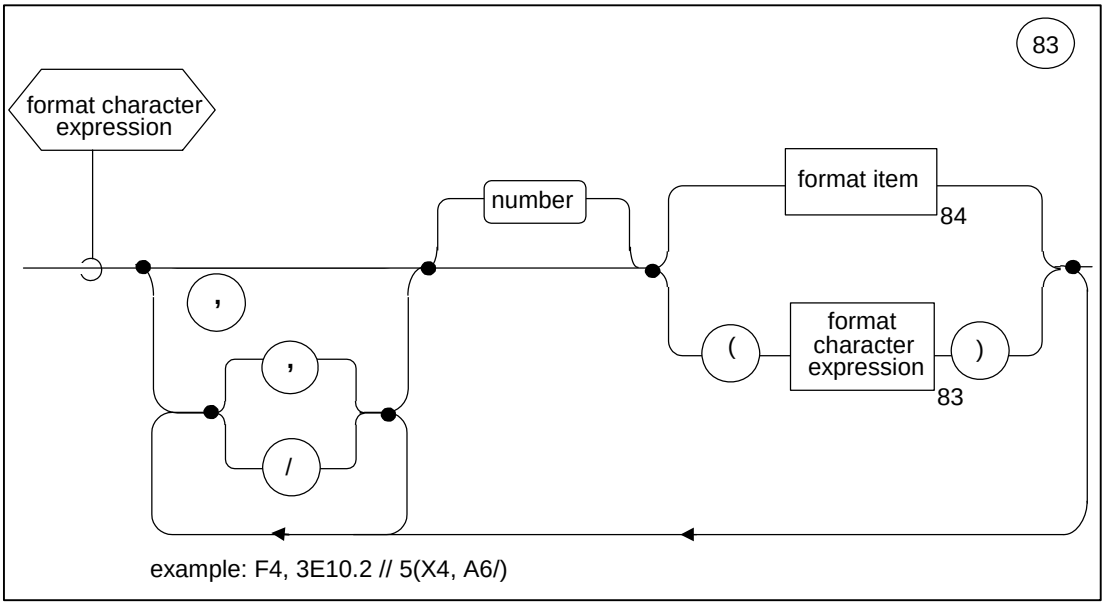


Figure 10-5 format character expression - #83

SEMANTIC RULES:

1. <number> must be an unsigned, non-negative integer.
2. Each item input or output is handled according to a particular format item. If <number> precedes a format item, it is interpreted as if <number> copies of the format item had been written. If <number> precedes a parenthesized <format character expression>, it is interpreted as if <number> copies of the <format character expression> had been written.

3. Each invocation of a READ or WRITE statement containing a <format list> interprets the <format character expression> starting from the beginning.
4. If the <format character expression> is exhausted and additional items remain to be input or output, control is returned to the <format character expression> corresponding to the last closed parenthesis encountered. A preceding <number> is taken into account if present. If no embedded <format character string>s are present, control reverts to the beginning.
5. '/' is interpreted as ',SKIP(1), COLUMN(1),'.
6. Consecutive commas are ignored.

Example:

```
DECLARE ARRAY(20), DIM VECTOR, ANIMAL CHARACTER(15);
WRITE(6) ('ANIMAL', 'LENGTH', 'WIDTH', 'HEIGHT')
    IN 'A15, (A10)';
DO FOR TEMPORARY I = 1 TO 20;
    WRITE(6) (ANIMAL_I, DIM_I,) IN 'A15, (F10.1)';
END;
```

produces:

```
          ANIMAL       LENGTH    WIDTH    HEIGHT
       CENTIPEDE          3.1       .3        .2
       AARDVARK          42.7     -12.6       8.2
             .  ↑          .↑        .↑        .↑
             .             .         .         .
             .             .         .         .
       column  15          25        35        45
```

10.1.4.2 FORMAT Items.

Each FORMAT item conceptually represents a single I/O operation.

SYNTAX:

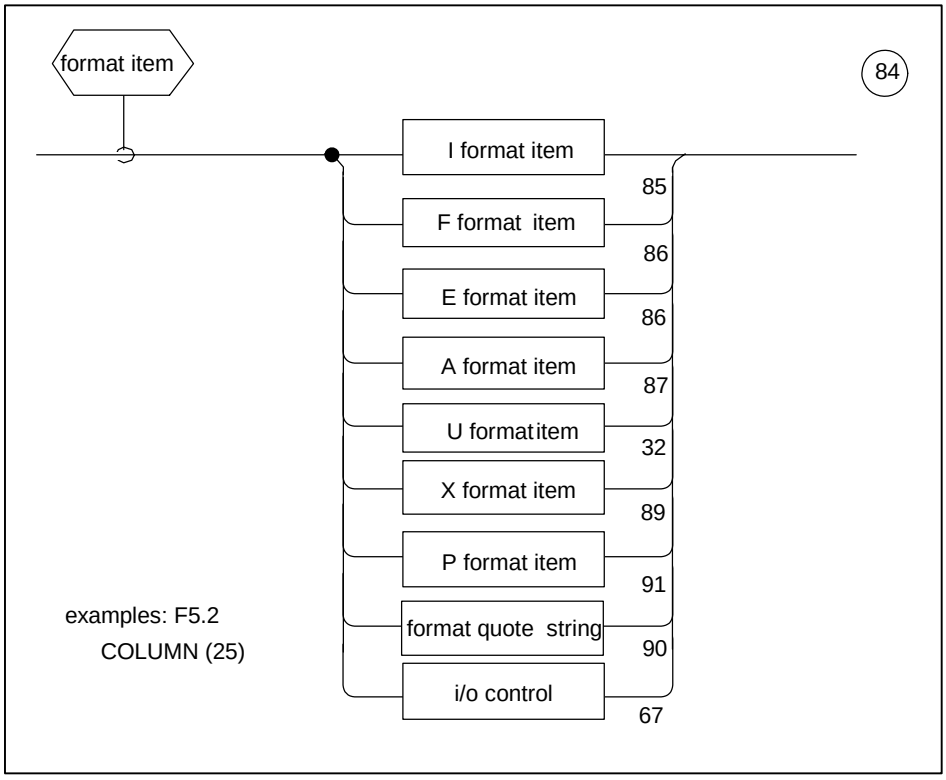


Figure 10-6 FORMAT item - #84

SEMANTIC RULES:

1. At the beginning of the READ or WRITE statement and after processing an item, X format items, quote strings, and I/O control are processed until some other format item is reached.
2. The semantics of <i/o control> were defined in Section 10.1.3.
3. The following table briefly describes the formats. See individual items for fuller applications.

ITEM	USE	EXAMPLE	SAMPLE OUTPUT	SAMPLE INPUT	INTERPRETED AS
I Format	INTEGER	I5	~~bbb~~ 97	~~bbb~~ 42	42
F Format	SCALAR	F6.2	~~b~~ 98.67	98.672	98.672
				~~bb~~ 9867	98.67
E Format	SCALAR with exponents	E9.1	~~b~~-7.1E-02	~~bb~~246E+14	24.6E+14
U Format	INTEGER, SCALAR, or CHARACTER	U5	~~bbb~~ 97	~~bbb~~ 42	42
A Format	CHARACTER	A4	~~b~~ABC	~~b~~ABC	~~b~~ABC
X Format	blanks on output, skips on input	X2	bb	9Z	skipped
P Format	INTEGER and SCALAR	PANS=$$.$*SS	ANS=-4.2E-8	-4.2E-8	-4.2E-8

10.1.4.3 I FORMAT Item

I FORMAT items are used for INTEGER I/O.

SYNTAX:

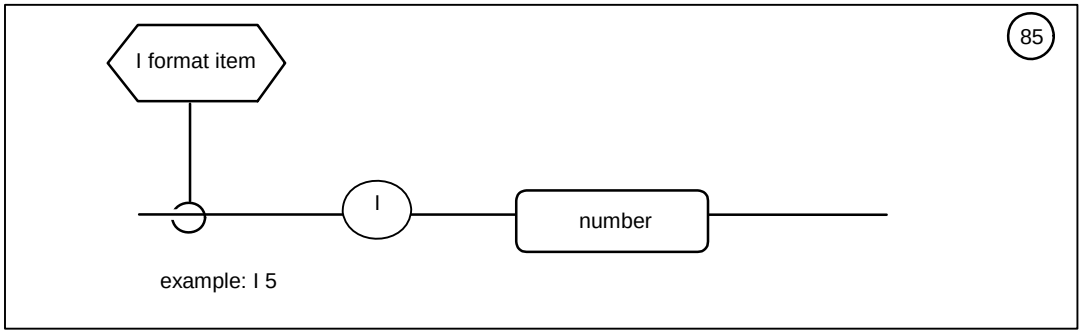


Figure 10-7 I FORMAT Item - #85

SEMANTIC RULES:

1. <number> is the length of the field being transmitted. It is an unsigned positive integer.
2. Implicit INTEGER/SCALAR conversion is allowed.

For READ statements:

1. A sign may precede the input quantity.
2. In input data, blanks before a sign or between a sign and the first digit are allowed. All other positions must contain digits between 0 and 9; otherwise, a runtime error occurs.

For WRITE statements:

1. A sign is printed only if a number is negative.

2. If the number of print positions required to represent the quantity is less than <number>, the leftmost positions are filled with blanks. If greater than <number>, a runtime error is sent and asterisks are printed in place of the quantity.

   Example:

```
DECLARE A INTEGER INITIAL (3);
WRITE(6) (A, -2, A-2) IN 'I2';
```

   produces:

   ~~b~~3-2~~b~~1

10.1.4.4 F and E FORMAT Items.

F FORMAT items are used for decimal quantities. E FORMAT items are used for decimal items written in scientific notation (i.e., with exponents).

SYNTAX:

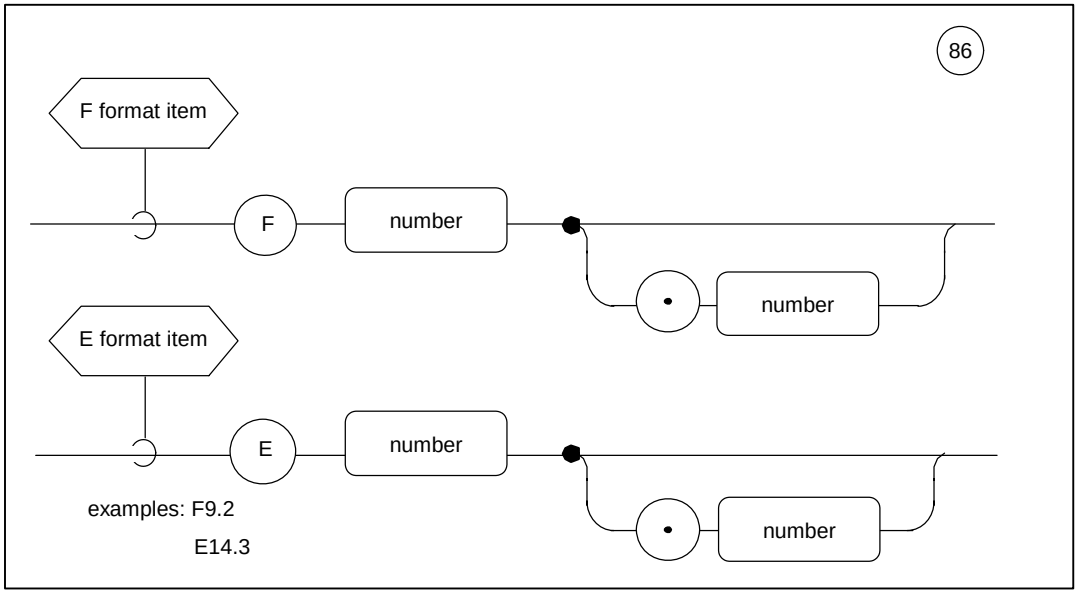


Figure 10-8 F and E FORMAT items - #86

SEMANTIC RULES:

1. The first <number> is the width of the field being transmitted. The optional second <number> specifies the number of decimal places to the right of the decimal point; if it is omitted, it is assumed to be zero. Each <number>, if present, is an unsigned positive integer.
2. Implicit INTEGER/SCALAR conversion is allowed.

For READ statements:

1. Input is an optionally signed quantity.
2. If an explicit decimal point appears in the input, it overrides the format; otherwise, decimal position is implied by the <format item>.

   Example:

   READ (5) (A,B,C) IN 'F6.3';

   interprets: ~~b~~12.34 as 12.34

   ~~bb~~1234 as 1.234

   ~~b~~.1234 as .1234
3. An exponent may be supplied in the form:

   E ±<number>

   If either E or ± is specified, the other may be omitted.
4. For input quantities, blanks are allowed preceding the sign, the first digit, E, ± , and the first digit of the exponent. Other blanks cause a runtime error.
5. There is no difference between E and F formats in the READ statements.

For WRITE statements:

1. For F format items, the string printed is:

   ```
   -aaaa.bbb
   └──┘ └┘
    m    n
   ```

   where n is determined by the second number in the format, and m is determined by the magnitude of the quantity to be printed. The minus sign is printed only if the quantity is negative. If the number of print positions required to represent the quantity is less than the field length, a zero is added to the left of the decimal if no other digits are present there. Any additional positions are filled with blanks from the left.
2. For E format items, the quantity printed is:

   ```
   -a.bbbE±cc
     └┘
      n
   ```

   The minus sign is printed only if the quantity is negative. One significant digit is printed to the left of the decimal point. This is zero if the quantity = 0. N is taken from the format item.
3. If the field length is insufficient, an error is sent and asterisks are printed in the field.

10.1.4.5 A FORMAT Items.

<A format item>s are used for character data.

SYNTAX:

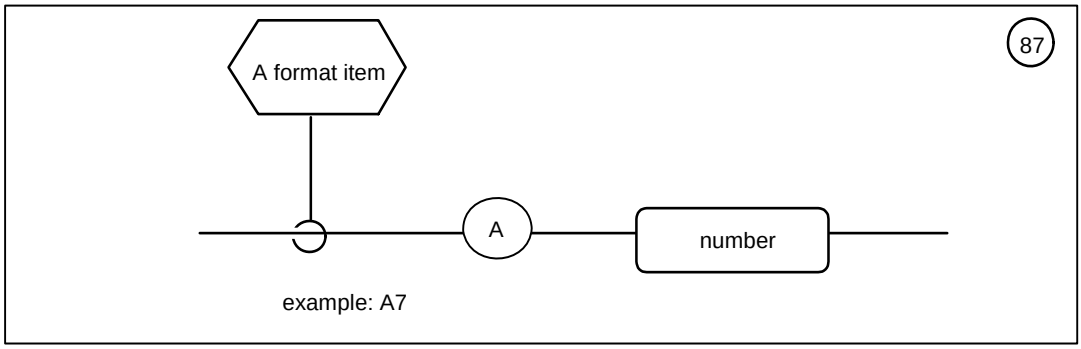


Figure 10-9 A format item - #87

SEMANTIC RULES:

1. <number> is an unsigned positive integer representing the field length.

For READ statements:

1. If the field specified is greater than the declared length of the variable, the rightmost characters in the field are selected. Otherwise, the length of the CHARACTER variable is set to the field length.

For WRITE statements:

1. If the field length written is greater than the number of characters in the variable, blanks are added to the left. Otherwise, the leftmost characters are written to fill the field.

Example:

```
WRITE (6) (PERSON_I, HEIGHT_I) IN 'A10, X2, F5.2';
```

would produce:

```
      BAGLEY   55.67
           ↑        ↑
columns   10       17
```

Note: BIT and CHARACTER conversion functions can be used with <A format item>s for I/O involving bit variables (see Sections 6.5.2 and 6.5.3).

10.1.4.6 U FORMAT Items.

<U format item>s are used for integer, scalar, and character data I/O.

SYNTAX:

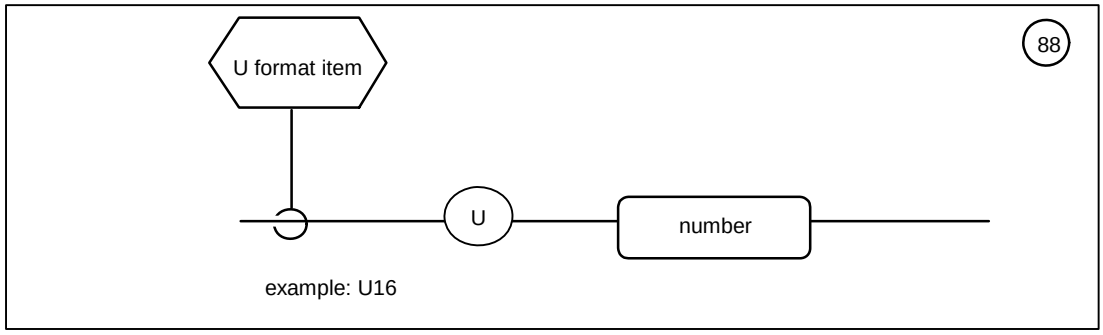


Figure 10-10 U format item - #88

SEMANTIC RULES:

1. <number> is an unsigned positive integer representing the field width.
2. The interpretation of the <U format item> depends upon the data type of the associated <variable> or <expression>.
   - for character strings, U<number> is equivalent to A<number>;
   - for integers, U<number> is equivalent to I<number>;
   - for scalars, U<number> is equivalent to E<number>.<number>-7.

10.1.4.7 X FORMAT Items.

<X format item>s are used to skip columns on input and output.

SYNTAX:

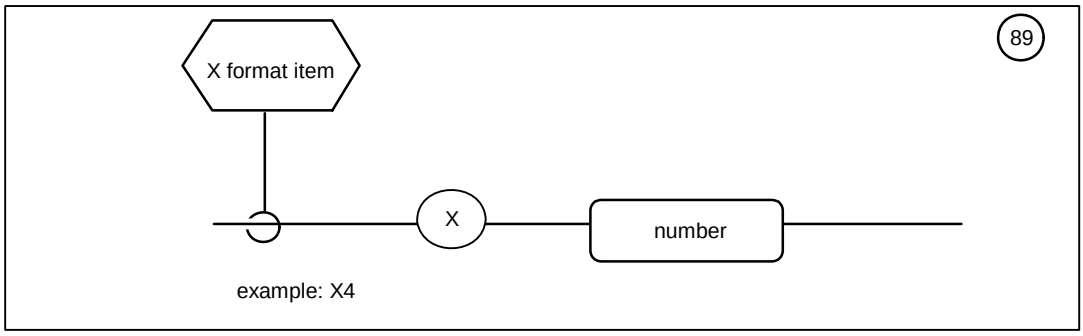


Figure 10-11 X format item - #89

SEMANTIC RULES:

1. <number> is an unsigned positive integer.
2. The effect is the same as TAB (<number>).

Example:

```
READ (5) A in 'X5, I3';
If the input is:
               12345678
then A becomes:
               678.
```

10.1.4.8 FORMAT Quote Strings.

FORMAT quote strings are used for character output.

SYNTAX:

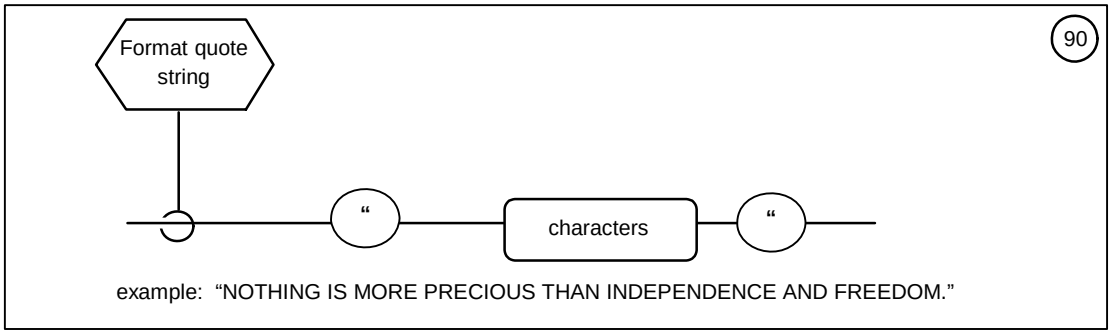


Figure 10-12 FORMAT quote string - #90

SEMANTIC RULES

For READ statements:

1. Columns corresponding to FORMAT quote strings are skipped in READ statements.

For WRITE statements:

1. A double quote in the text is represented by a pair of double quotes.
2. <character>s are copied to the output line.

Example:

```
WRITE(6) ANS IN '"ANSWER = ", I2';
```

would produce:

```
          ANSWER = 21
                 ↑
columns   1
```

10.1.4.9 P FORMAT Items.

<P format item>s can be employed for most types of numeric I/O. They can be very useful in mixing character and numeric output data and specifying column alignment.

SYNTAX:

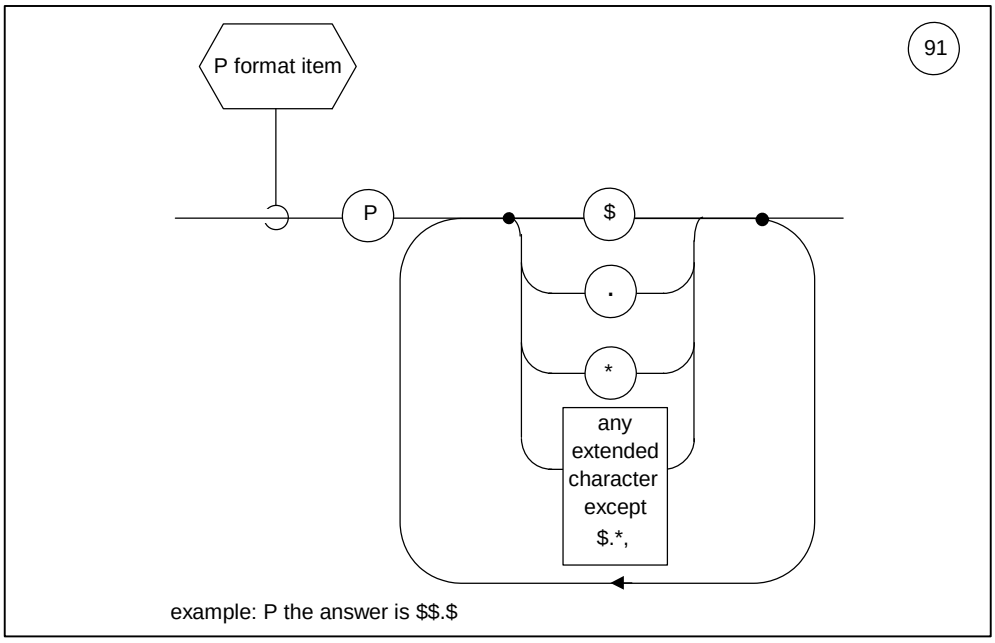


Figure 10-13 P format item - #91

SEMANTIC RULES:

1. The <P format item> runs from the first character following the P to the first ‘,’ or ‘/’ encountered (or the end of the format character string).
2. Each set of consecutive ‘$’s, ‘.’s, and ‘*’s defines a numeric field corresponding to an INTEGER or SCALAR item. ‘*’ defines the beginning of an exponent. If more than one ‘.’ or ’*’ is present in a given numeric field, a runtime error is sent.
3. More than one field is allowed, e.g.,

```
WRITE(6)(NO,ARG1,ARG2,ARG1+ARG2) IN 'P TEST#$$:$.$$*$$$+$.$$*$$=$.$$*$$$';
```

For READ statements:

1. Each field length is the number of ‘$’, ‘.’, and ‘*’ present. Other characters cause corresponding columns to be skipped.
2. A decimal in the input field takes precedence. Otherwise, a decimal is placed by the ‘.’, if present.
3. An exponent may be supplied of the form:

   E ± <number>

   If either E or ± is specified, the other may be omitted.

4. Blanks are allowed preceding the sign, the first digit, E, ±, and the first digit of the exponent. Other blanks cause a runtime error.

   Example:

   ```
   READ(5) (X,Y) IN 'PXXX$$.$$X$$';
   ```

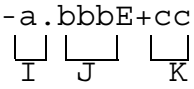
   then if the input is:

   ```
   01234567890
   ```

   then X would be set to 345.67 and Y to 90.

For WRITE statements:

1. If a quantity to be printed is smaller than the specified field width, blanks are appended to the left. If the quantity to be printed (including '-' if needed) is larger than the specified field width field, a runtime error is sent and the first is filled with asterisks.
2. All characters except '$', '*', and ',' are printed.
3. If an exponent is called for, the number takes the form:

   A non-negative quantity prints a blank in place of the '-' sign. The leftmost digit printed will be non-zero unless the value to be printed is exactly zero. The field widths I, J, and K are taken from the number of '$' signs in the picture. I must be greater than zero and K large enough to hold the exponent.

   Example:

   ```
   DECLARE CHARACTER(100),
   T INITIAL('P TITLE1   TITLE2   TITLE3/'),
   D INITIAL('P  $$.$$    $$.$$    $$.$$/')
   WRITE(6) IN T;
   WRITE(6) DATA ARRAY$(1 TO 9) IN D;
   ```

   would produce the following table:

   ```
   TITLE1    TITLE2    TITLE3
    98.72    -5.61      43.00
   ```

   ```
       ↑         ↑         ↑
   column   8         17         26
   ```

10.2 Random Access I/O and the FILE Statement.

Random access I/O is handled by means of the FILE statement. In this access method individual records on a file may be written, retrieved, or updated. A unique “record address” is used to specify the particular record on the file referenced.

SYNTAX:

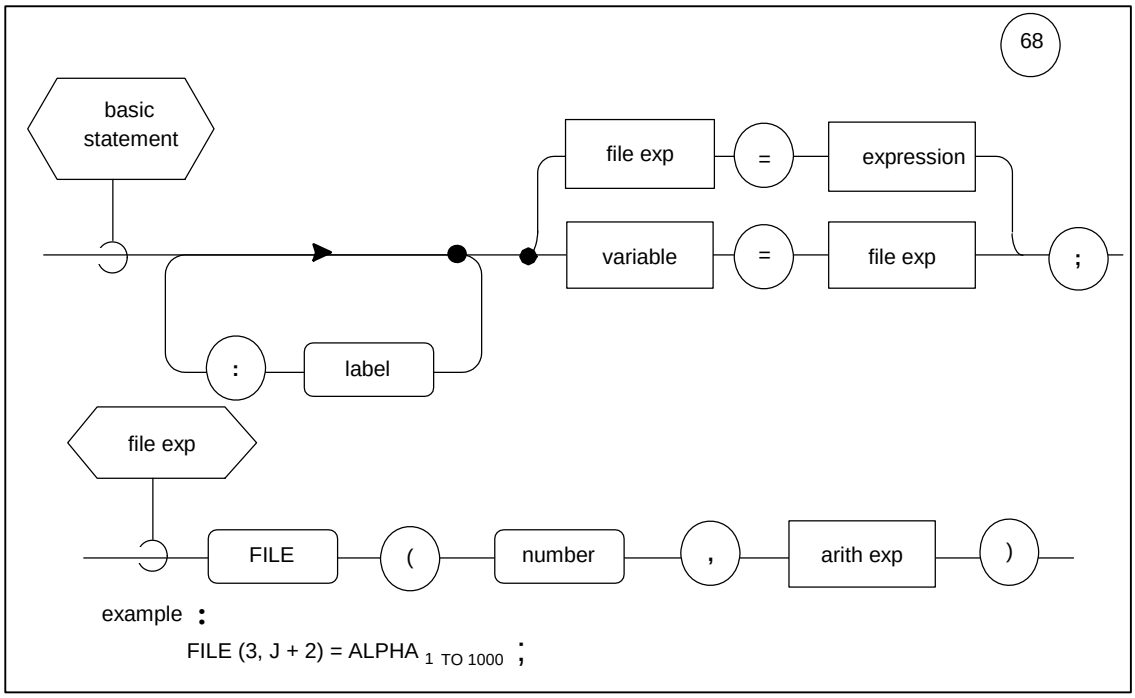


Figure 10-14 FILE statements - #68

SEMANTIC RULES:

1. The statement is an output FILE statement if <file exp> is on the left of the assignment. If <file exp> is on the right, then the statement is an input FILE statement.
2. <file exp> specifies the random access I/O channel and record address to be referenced. <number> is any legal random access channel number. <arith exp> is any unarrayed integer or scalar expression. If the expression is scalar, its value is rounded to the nearest integer before use. A runtime error occurs if its value is not a legal record address.
3. Any record on a random access file may be transmitted by a FILE statement.
4. In the input FILE statement, <variable> is any variable usable in an assignment context. This specifically excludes input parameters of function and procedure blocks. Moreover, <variable> is also subject to the following rules:
   - No component subscripting for bit and character types.
   - If component subscripting is present, <variable> must be unsubscripted so as to yield a single (unarrayed) element of the <variable>.

- If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.
- BIT type structure terminals which have the DENSE attribute may not be used, due to packing implications. However, an entire structure with the DENSE attribute may be used.
- If the <variable> is a structure terminal or a minor structure node (but not a major structure) and if the structure possesses multiple copies, then the number of copies must be reduced to one by subscripting.

5. In the output FILE statement, there are no semantic restrictions on <expression>.
6. Compatibility between data written by an output FILE statement, and later reference to it by an input FILE statement is assumed. The exact interpretation of compatibility is implementation dependent. In general, the FILE statement transmits binary images of the internal data forms, so that compatibility will be guaranteed if the <expression> of the output FILE statement and the <variable> of the input FILE statement have the same data type and organization.

11.0 SYSTEMS LANGUAGE FEATURES[17]

11.1 INTRODUCTION.

The systems language features of HAL/S are described in this section. The features presented here are in three sections. The new Program Organization features are “Inline Function Blocks” and “%macros”. A data-related feature of this systems language extension is the concept of “TEMPORARY variables”. The NAME facility concerns a new concept in HAL/S, the addition of NAME variables pointing to data or blocks of code.

The information contained in this section constitutes an extension of material presented earlier. Accordingly, many of the syntax diagrams presented here are modified versions of earlier diagrams reflecting the extended features. Such modified diagrams are indicated by appending the small letter “s” to the diagram number.

11.2 PROGRAM ORGANIZATION FEATURES

The addition of Inline Function Blocks and “%macros” to HAL/S extends the information presented in Section 3 concerning program organization. Inline functions are a modified kind of user function in which invocation is simultaneous with block definition. %macros may be viewed as a class of special purpose implementation dependent built-in functions.

11.2.1 Inline Function Blocks.

The HAL/S Inline Function Block is a method of simultaneously defining and invoking a restricted version of the ordinary user function construct. Its primary purpose is to widen the utility of the parametric REPLACE statement described in Section 4.2. Its appearance is generally in the form of an operand of an expression.

An Inline Function Block, like other blocks, has a new level of name scoping and error recovery.

17. The title indicates that the usage of these constructs is more suited to systems programming rather than applications programming. The programmer is warned that unrestrained and indiscriminate use of certain of these constructs can lead to software unreliability.

SYNTAX:

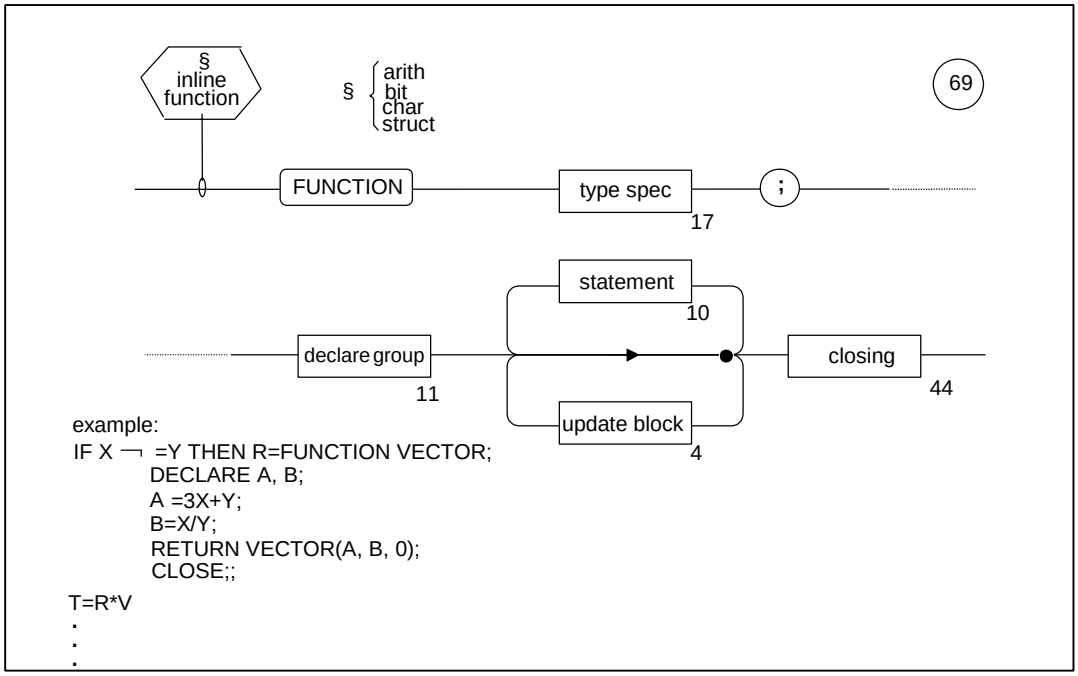


Figure 11-1 Inline Function Block - #69

SEMANTIC RULES:

1. The syntactical form is actually equivalent to that of a function block except that:
   a The <§inline function> has no label;
   b The <§inline function> has no parameters;
   c The <§inline function> definition becomes an operand in an expression.
2. The semantic rules for an <§inline function> block definition are the same as those for the <function block> definition described in Section 3.3, subject to the restrictions listed below.
3. A <§inline function> may not contain the following syntactical forms:
   - All forms of I/O statements;
   - All forms of reference to user-defined PROCEDURE and FUNCTION blocks;
   - Real Time statements;
4. A <§inline function> may only contain one form of nested block, the <update block>. The following block forms are thus excluded:
   - <function block> definitions;
   - <procedure block> definitions;
   - Further nested <§inline function>s.
5. In use, the following semantic restriction holds: <§inline function>s may not appear as operands of subscript or exponent expressions.

6. The <§inline function> falls into one of the following four categories:

<arith inline>	-	<type spec> specifies an inline function of an arithmetic data type: SCALAR, INTEGER, VECTOR, or MATRIX.
<bit inline>	-	<type spec> specifies an inline function of a bit type: BOOLEAN or BIT.
<char inline>	-	<type spec> specifies an inline function of the CHARACTER data type.
<struct inline>	-	<type spec> specifies an inline function with a structure type specification.

The use of inline functions as operands of HAL/S expressions is discussed in Section 11.2.3.

11.2.2 %macro References.

The HAL/S %macro facility provides a means of adding functional, special-purpose extensions to the language without requiring syntax changes or extensive rewriting of the compiler programs. The details of the implementation of any given %macro will depend upon its nature and purpose. Possible options include inline generation of code or links to an external routine performing the processing of the %macro.

The syntax of %macro reference is presented in this section. The invocation of %macro routines in various expression or statement contexts is described below in Sections 11.2.3 and 11.2.4.

SYNTAX:

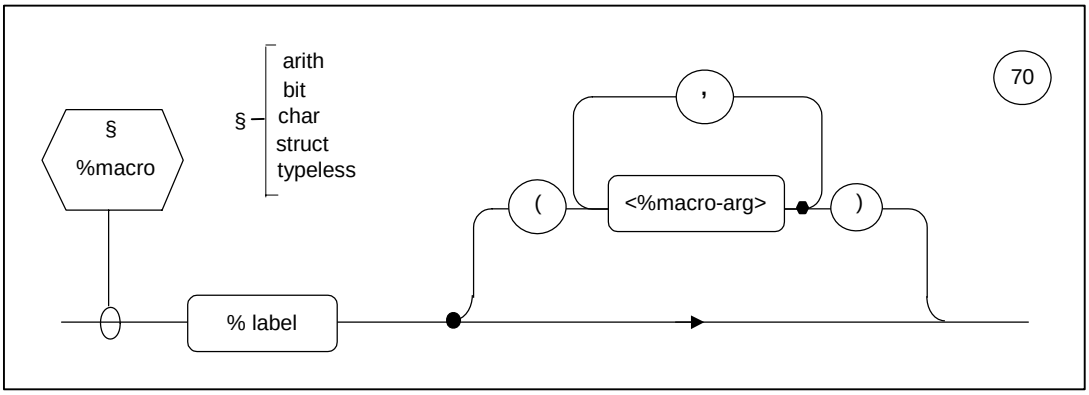


Figure 11-2 %Macro Statement - #70

SEMANTIC RULES:

1. The §macro reference falls into one of the five following categories based upon data type:
   - <arith %macro> is a reference to a §macro which returns an arithmetic value of INTEGER, SCALAR, VECTOR, or MATRIX data type.
   - <bit %macro> is a reference to a §macro which returns a bit string value.
   - <char %macro> is a reference to a §macro which returns a value of CHARACTER data type.
   - <struct %macro> is a reference to a §macro which returns a structure data value.

- <typeless %macro> is a reference to a §macro which performs some systems function but returns no value and may only be referenced from a §macro call statement (see Section 11.2.4 below).

Available %macros in any implementation will be provided in the appropriate User's Manual.

2. The <%label> is a reserved word beginning with the character "%" which identifies the %macro in question. The character "%" distinguishes %macro names from all other reserved words in the HAL/S language.
3. A series of one or more arguments of the %macro reference may be supplied. The type, organization, and number of the arguments supplied to the %macro must be consistent with the requirements of the routine.
4. Details of <%macro arg>s will be supplied with the definition of a given %macro.

11.2.3 Operand Reference Invocations.

Inline Function Blocks are always invoked at the point of their definition as operands of <expression>s. %macros are also invoked as operands of <expression>s when they are of a definite data type and thus return a value. Similar modifications of several syntax diagrams from Section 6 add these features to arithmetic, bit, and character operands, and to structure expressions.

SYNTAX OF ARITHMETIC OPERANDS:

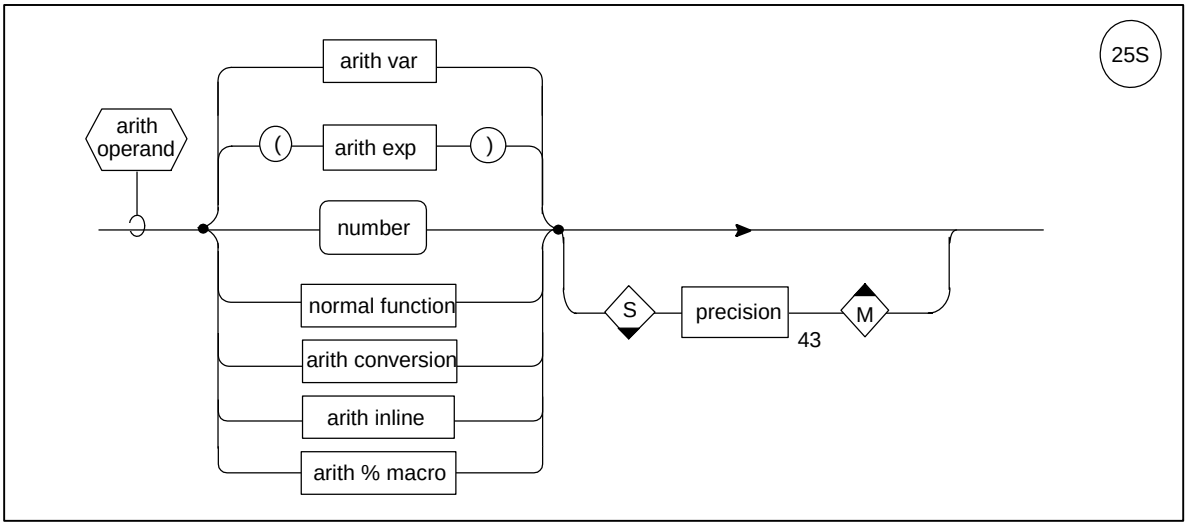


Figure 11-3 arithmetic operand - #25s

SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the arithmetic operand diagram in Section 6.1.1. The semantic rules of Section 6.1.1 apply to this revised diagram.
2. <arith inline> is an inline function block which has an arithmetic <type spec> in its header statement.
3. <arith %macro> is a reference to a %macro which returns an arithmetic value (see Section 11.2.2 above).

SYNTAX OF BIT OPERAND:

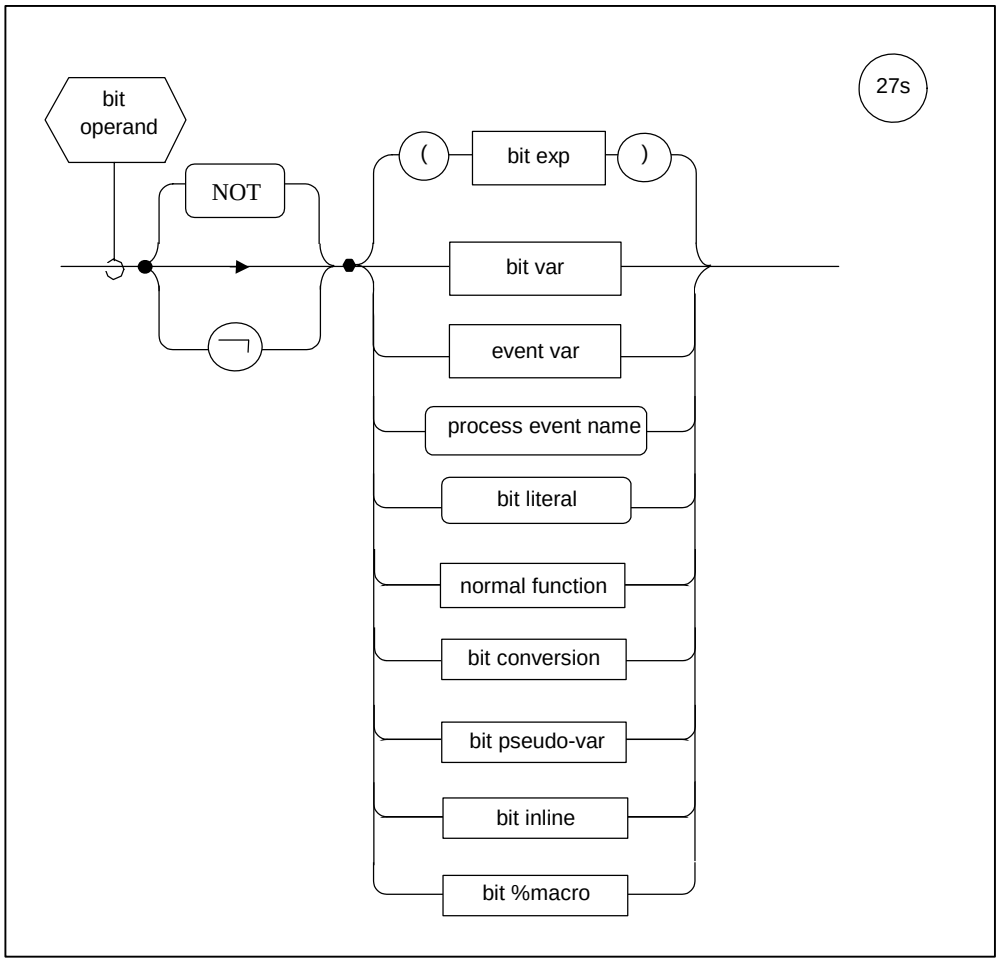


Figure 11-4 bit operand - #27s

SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the bit operand diagram in Section 6.1.2. The corresponding semantic rules found in Section 6.1.2 also apply to this revised diagram.
2. <bit inline> is an inline function block which has a bit string (BOOLEAN or BIT) <type spec> in its header statement.
3. <bit %macro> is a reference to a %macro which returns a value of the BIT or BOOLEAN data types.

SYNTAX OF CHARACTER OPERAND:

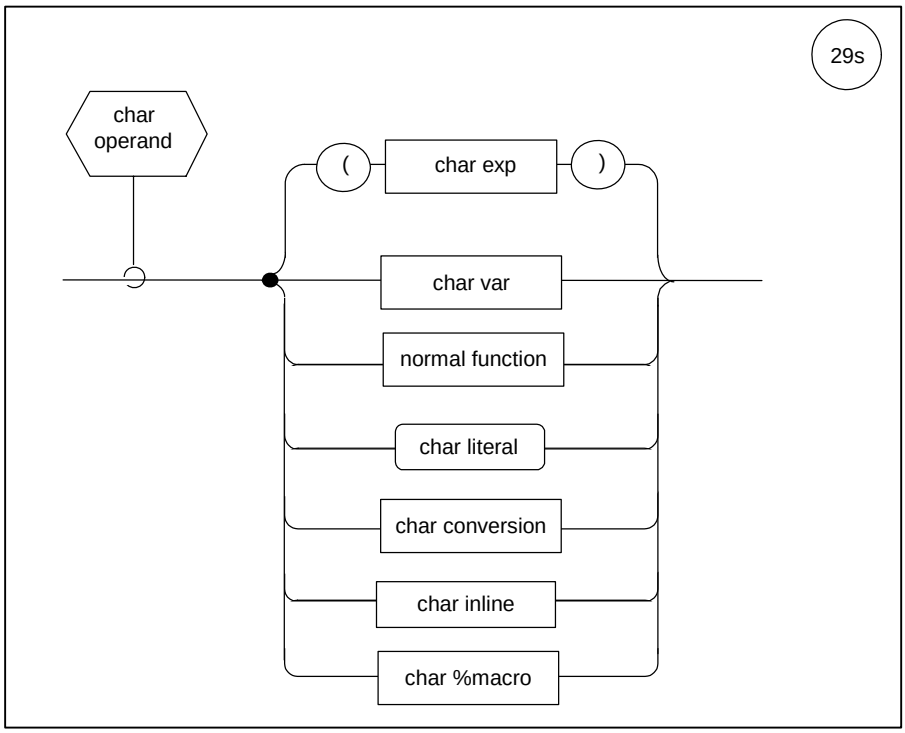


Figure 11-5 character operand - #29s

SEMANTIC RULES:

1. The syntax diagram is a systems language extension of the character operand diagram in Section 6.1.3. The corresponding semantic rules found in Section 6.1.3 also apply to this revised diagram.
2. <char inline> is an inline function block which has a CHARACTER <type spec> in its header statement.
3. <char %macro> is a reference to a %macro which returns a value of the CHARACTER data type.

SYNTAX OF STRUCTURE EXPRESSIONS:

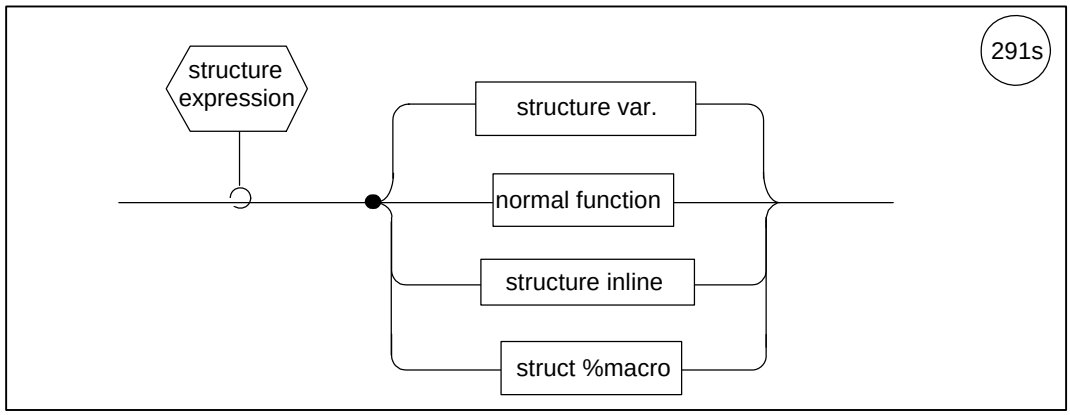


Figure 11-6 structure expression - #29.1s

SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the structure expression diagram found in Section 6.1.4. The semantic rules found in Section 6.1.4 also apply to this revised diagram.
2. <struct inline> is an inline function block which has a structure <type spec> in its header statement.
3. <struct %macro> is a reference to a %macro which returns a value of a structure data type.

11.2.4 The %Macro Call Statement.

The invocation of a typeless %macro is performed by a <%macro call statement>.

SYNTAX:

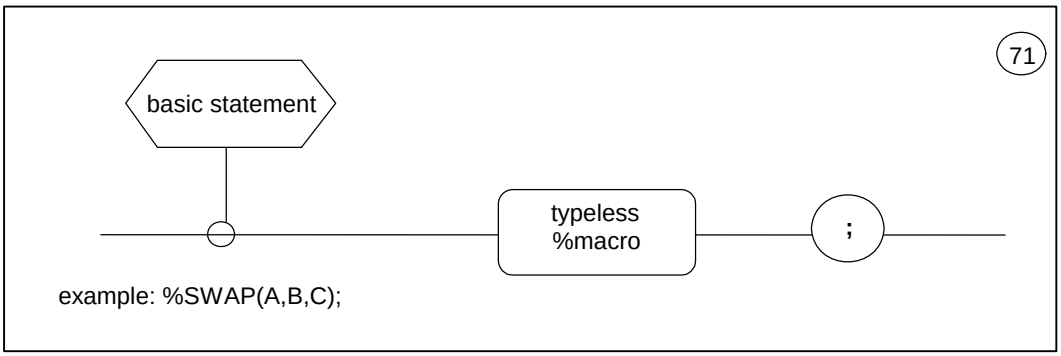


Figure 11-7 %MACRO - #71

SEMANTIC RULES:

1. The <%macro call statement> invokes execution of the typeless %macro being referenced.
2. The effect of this statement is dependent upon the details of the %macro being referenced.

11.3 Temporary Variables.

The extension of HAL/S data concepts to include a TEMPORARY variable form for use within DO groups is defined within the systems language facilities. The object of incorporating the TEMPORARY variable is to increase the optimization and efficiency of the object code produced by the compiler. Depending upon the details of the object machine, a TEMPORARY variable might be stored in a CPU register or a high speed, scratch pad memory location rather than in the slower main storage. Coding efficiency may also be achieved with TEMPORARY variables because the instructions needed to access register or scratch pad memory values are generally more compact. Since the existence of a TEMPORARY variable is confined to a DO group (from DO header statement to the END statement), these forms become highly localized control variables.

If a TEMPORARY variable appears in a REENTRANT block, each process simultaneously executing the block gets its own TEMPORARY variable.

11.3.1 Regular TEMPORARY Variables.

Regular TEMPORARY variables are declared in TEMPORARY statements following the DO statement which begins a DO…END statement group and preceding the first executable statement of the DO…END statement group. The following diagram is a systems language extension of the DO…END statement group in Section 7.6.

SYNTAX:

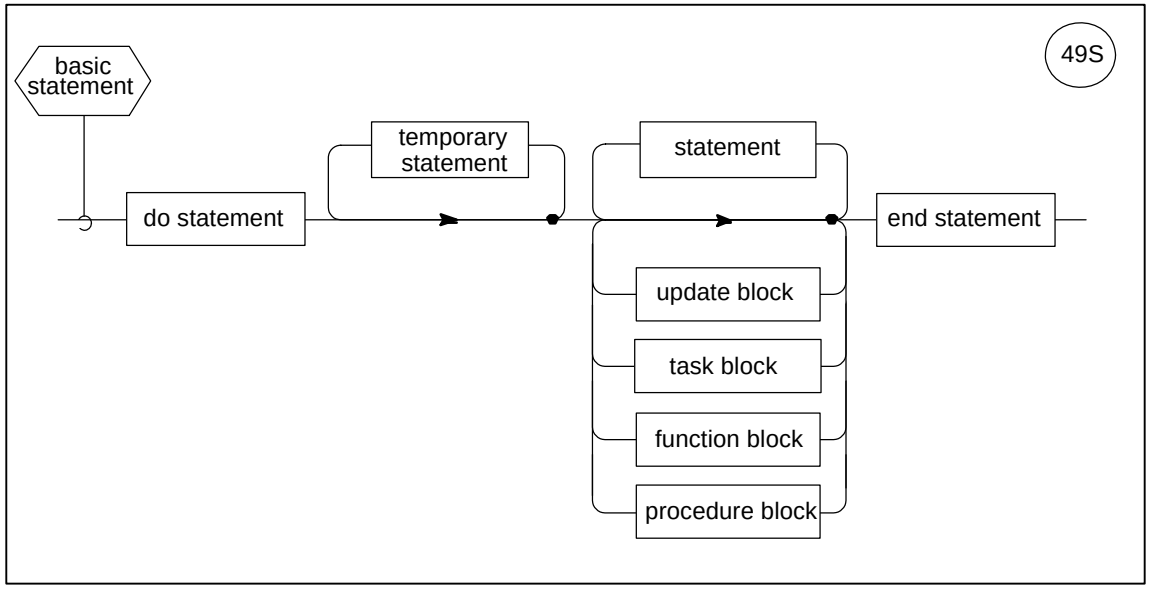


Figure 11-8 DO…END Statement Group - #49s

SEMANTIC RULE:

1. The TEMPORARY declaration may be included as part of any DO group except a DO CASE group. Use of TEMPORARY variables within nested DO groups of a DO CASE is allowed.

The TEMPORARY statement is a special purpose data declaration used to create TEMPORARY variables for general use within the DO group syntax as described above. Its form compares very closely to that of the DECLARE statement in Section 4.4.

SYNTAX:

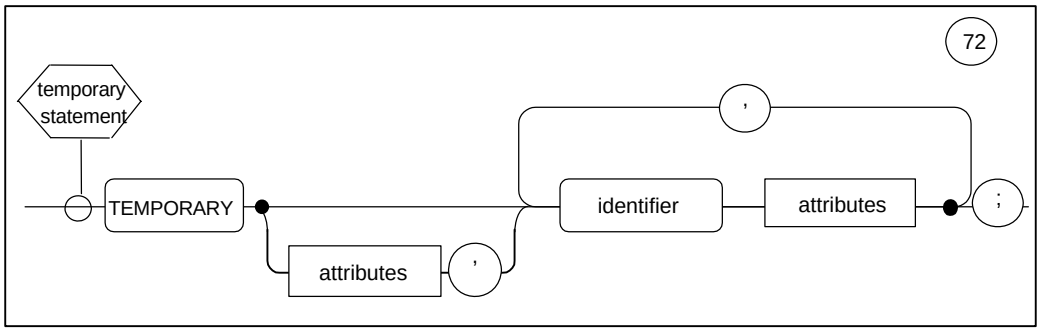


Figure 11-9 temporary statement - #72

SEMANTIC RULES:

1. In the <temporary statement>, <attributes> may define the <identifier>s to be of any data type except EVENT.
2. <attributes> may only specify type, precision, and arrayness.
3. No minor attribute is legal.
4. The name of <identifier> may not duplicate the name of another <identifier> in the same name scope (procedure, function, or other block name) or of another temporary in the same DO…END group.

11.3.2 Loop TEMPORARY Variables.

The Loop TEMPORARY variable form is used in the context of the DO FOR group and is declared by its specification in a DO FOR statement. The following two syntax diagrams are modifications of the discrete DO FOR and the iterative DO FOR syntax diagrams.

SYNTAX:

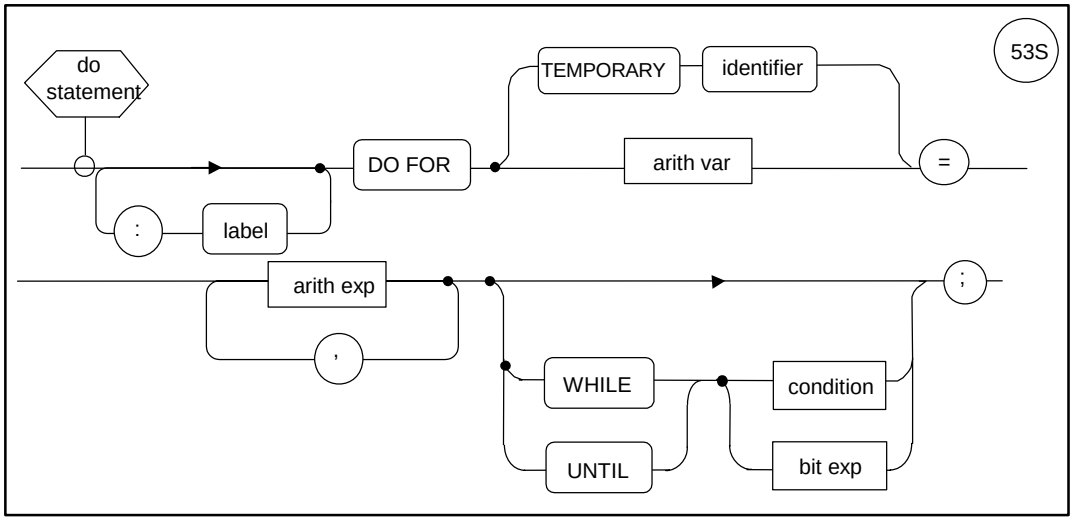


Figure 11-10 discrete DO FOR with loop TEMPORARY variable index - #53

SYNTAX:

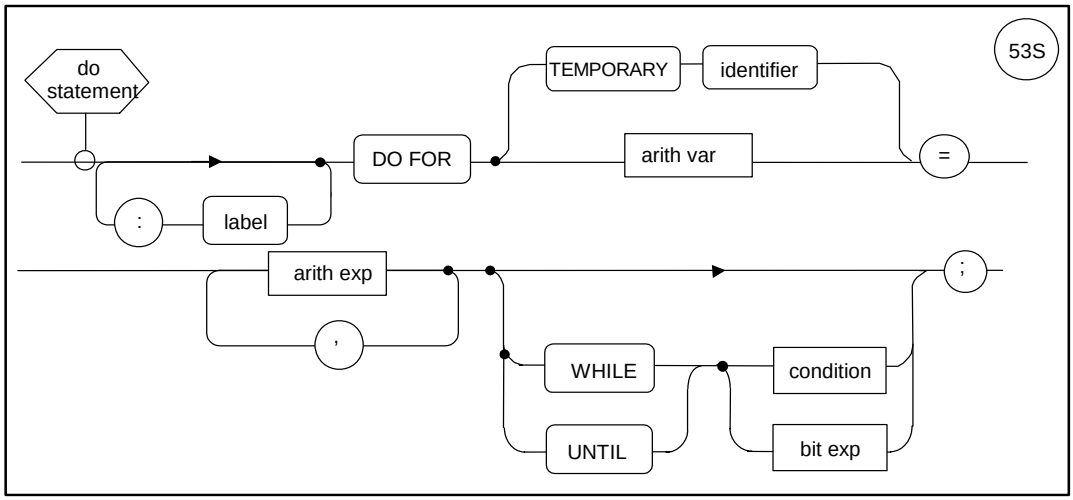


Figure 11-11 iterative DO FOR with loop TEMPORARY variable index - #54s

SEMANTIC RULES:

1. All the semantic rules for DO FOR statements which are given in Sections 7.6.4 and 7.6.5 apply as well to the corresponding Loop TEMPORARY forms. Additional rules for Loop TEMPORARY variables are given below.
2. The Loop TEMPORARY variable is defined in the DO FOR statement; a Loop TEMPORARY variable is always a single precision INTEGER variable.
3. The scope of the Loop TEMPORARY is the DO FOR group of the DO FOR statement which defines the variable.
4. The <identifier> name used for the Loop TEMPORARY may not duplicate the name of another <identifier> in the same name scope, nor may it duplicate the name of another TEMPORARY variable in the same DO…END group.

11.4 The NAME Facility

This section gives a definitive description of the HAL/S NAME facility. This facility is designed to fill the system programmer's need for a "pointer" construct. Its basic entity is the NAME identifier: a NAME identifier "points to" an ordinary HAL/S identifier of like attributes. The "value" of the NAME identifier is thus the location of the identifier pointed to (an ordinary identifier is a HAL/S identifier without the NAME attribute).

11.4.1 Identifiers with the NAME Attribute

Identifiers declared with the NAME attribute become NAME identifiers. NAME identifiers may be declared with the following data types:

```
INTEGER          CHARACTER
SCALAR           EVENT
VECTOR           STRUCTURE
MATRIX           PROGRAM
BIT              TASK
BOOLEAN
```

The following diagram is an extension of the DECLARE statement syntax diagram in Section 4.4. The modification shows how the keyword NAME is used in such a declaration to state the NAME attribute.

SYNTAX:

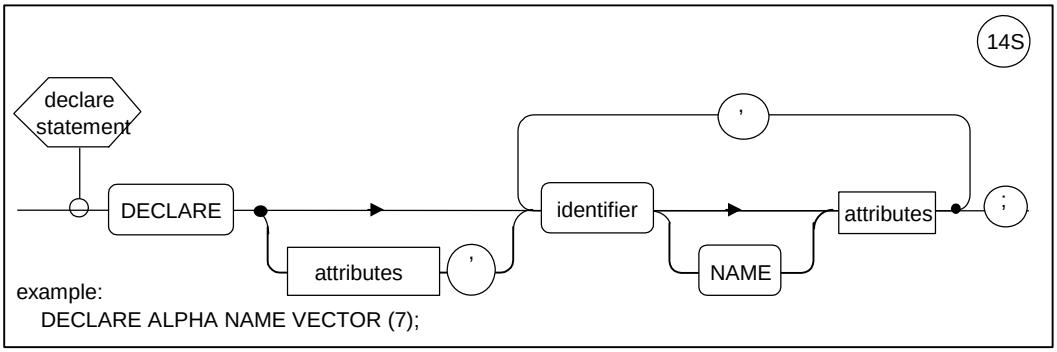


Figure 11-12 declaration statement - #14s

GENERAL SEMANTIC RULES:

1. The following <attributes> apply to the NAME variable itself and bear no relationship to the ordinary identifier which is pointed to at any given time during execution:
   - The <initialization> attribute (if supplied) refers to the initial pointer value of the NAME variable itself.
   - STATIC/AUTOMATIC refer to the mode of initialization of the NAME variable itself on entry into a HAL/S block.
   - DENSE/ALIGNED apply apply to the actual NAME variable when it is defined by inclusion in a structure template.

   All other legal attributes describe the characteristics of the ordinary variables to which the NAME variable may point. Except as noted below, these other attributes must always match the corresponding attributes of the ordinary variables to which the NAME variable points; compilation errors will ensue if this is not the case.
2. The ACCESS attribute is illegal for NAME variables; its absence does not prevent NAME identifiers from pointing to ordinary identifiers with the ACCESS attributes, and matching is not required in this case.
3. The REMOTE attribute need not match if the NAME variable is REMOTE. NAME REMOTE variables have a larger addressing range than non-REMOTE NAME variables and can point to either REMOTE or non-REMOTE variables; however, non-REMOTE NAME variables can only point to non-REMOTE variables.
4. There must still be consistency between declared type, attributes, and factored attributes just as is the case for ordinary identifiers as described in Chapter 4 of the Specification.

Examples:

```
DECLARE VECTOR(3) DOUBLE LOCK(2), X, Y NAME;
DECLARE P NAME TASK;
```

Y may point to X

P points to any task block

Figure 11-13 NAME Examples

SEMANTIC RULES (Data NAME Identifiers):

1. Arrayness Specification - in general, the arrayness specification of a NAME identifier must match that of the ordinary identifiers pointed to, in both number and size of dimensions.
2. Structure Copy Specification - in general, the number of copies of a NAME identifier of a structure type must match that of the ordinary identifiers pointed to.
3. The use of the "*" array specification or structure copies specification is excluded from declarations of NAME formal parameters.
4. Structure Type - if a NAME identifier is a structure type it may only point to ordinary identifiers of structure type with tree equivalent structure templates.

Examples of data NAME variables

```
DECLARE X ARRAY(3) SCALAR,
        Y ARRAY(4),
        Z NAME ARRAY(4) SCALAR;
DECLARE P EVENT;
DECLARE EVENT LATCHED, V, VV NAME;
```

Z may point to Y but not X

Figure 11-14 NAME Array Examples

5. For any unarrayed character string name variable, the "*" form of maximum length specification may be used. This is an extension of the use of the "*" notation which applies now in general to character name variables as well as to formal parameters.

The Label Declarative Attributes available for use in declaring NAME identifiers which point to HAL/S block forms have been modified to include PROGRAM and TASK keywords and to exclude PROCEDURE and FUNCTION keywords. The following syntax diagram is substituted for the Label Declarative Attributes diagram in Section 4.6 when declaring NAME identifiers which point to HAL/S blocks.

SYNTAX:

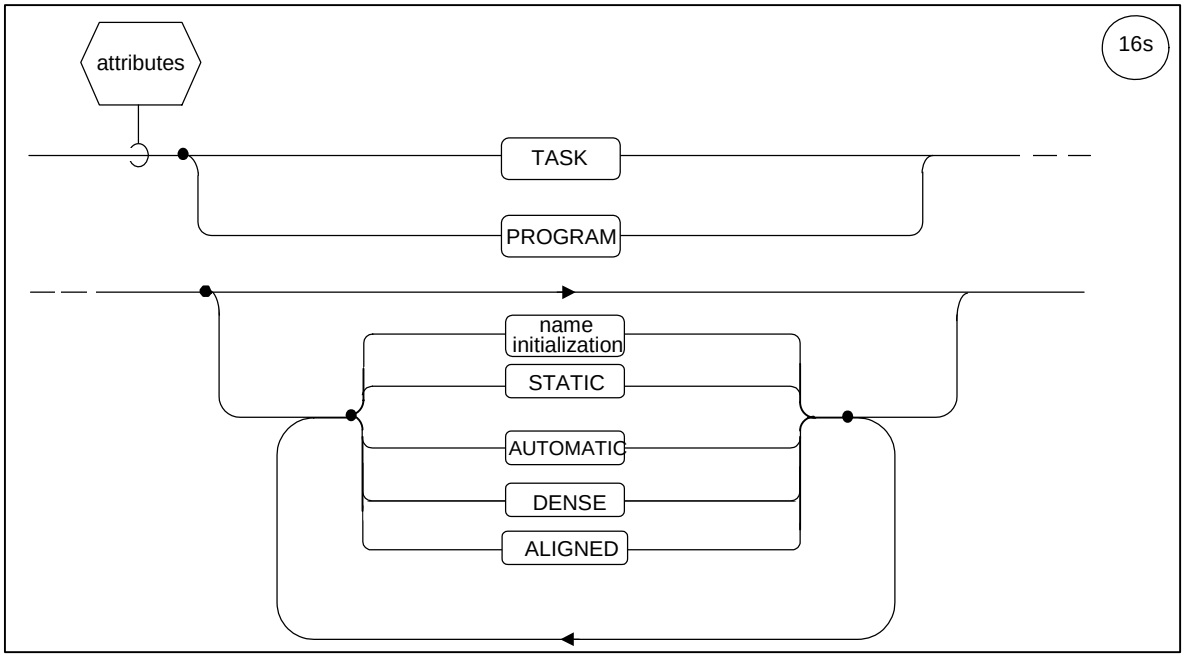


Figure 11-15 label declarative attribute - #16s

SEMANTIC RULES (Label NAME Identifiers):

1. <initialization>, STATIC or AUTOMATIC, DENSE or ALIGNED may only be applied to the <label declarative attributes> of identifiers with the NAME attribute. They are properties of the NAME and not of the identifiers pointed to.

2. The following rules apply to NAME <identifiers> of the PROGRAM and TASK types:

   - The NAME <identifier> of a PROGRAM or TASK type always points to a PROGRAM or TASK block, respectively. A corollary of this rule is that <process event>s are never referenced by NAME identifiers of the PROGRAM or TASK types.
   - The only forms of PROGRAM label declarations allowed are those with the NAME attribute.
   - The program NAME <identifier> must always point to an external PROGRAM block name; therefore, a block template is required for each PROGRAM which may be referenced by a NAME value.

11.4.2 The NAME Attribute in Structure Templates.

The NAME attribute may appear on any structure terminal of a structure template. The following syntax diagram shows how the keyword NAME is used to state the NAME attribute. This diagram is a systems language extension of the structure template diagram.

SYNTAX:

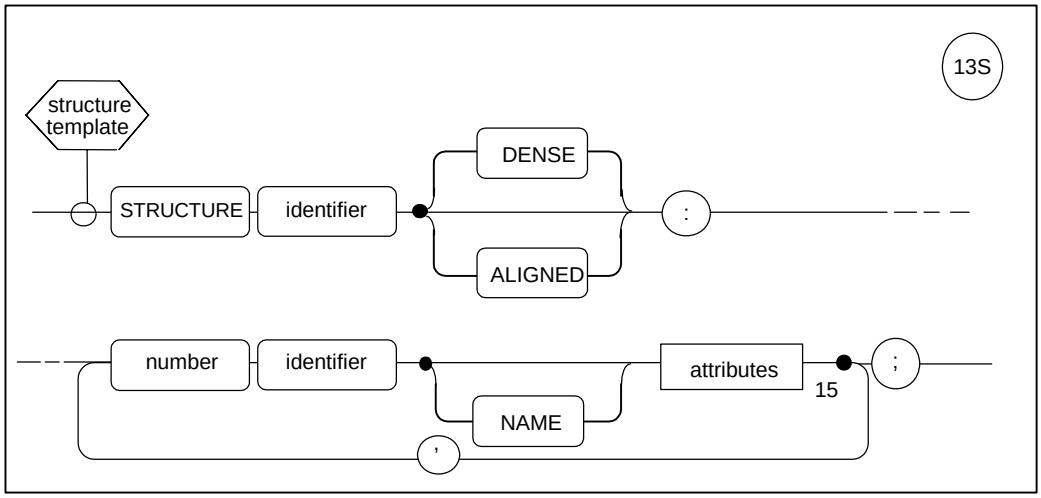


Figure 11-16 structure template statement - #13s

In general, the rules governing the formation of the structure template remain unchanged (see Section 4.3).

GENERAL SEMANTIC RULES:

1. Restrictions on attributes discussed in Section 11.4.1 generally also apply to structure terminals with the NAME attribute.
2. No <initialization> may be applied to terminals; neither may the attributes STATIC /AUTOMATIC appear.
3. NAME identifiers of any type (including program or task) may appear as structure terminals. Note that the NAME of an EVENT may appear in a structure even though the EVENT itself may not.
4. The REMOTE attribute may be applied to a structure terminal with the NAME attribute unless it is of EVENT type.

SEMANTIC RULES (Nested Structure Template References):

1. Nested structure template references are special instances of structure terminals. Their manner of incorporation into structure template definitions is described in Section 4.3 via the <type spec>.
2. Such references are permitted to use the NAME attribute. If the NAME attribute is present, the following points are to be noted:
   - Specification of multiple copies is still not permitted.
   - The reference may be to the structure template being defined (and of which the reference is a part). The implications of this are discussed later.

SYNTAX:

examples of structure NAME identifiers:

```
STRUCTURE A:
        1 X NAME PROGRAM,
        1 Y SCALAR,
        1 Z NAME SCALAR,
        1 ALPHA NAME A-STRUCTURE;
DECLARE P A-STRUCTURE;
DECLARE PP NAME A-STRUCTURE;
```

P.Z	is a NAME identifier which may point to P.Y
PP	is a NAME identifier which may point to P
PP.Z	is a NAME identifier which may point to P.Z which is itself a NAME identifier pointing somewhere. This is an instance of double indirection.
P.ALPHA	is a NAME identifier of A-structure type. The consequences of this are discussed later.

Figure 11-17 Structure NAME Examples

11.4.3 Declarations of Temporaries.

TEMPORARY data may possess the NAME attribute. Such data follows the same semantic rules as regular TEMPORARY variables as described in Section 11.3.1, except that the REMOTE attribute is legal for TEMPORARY NAME variables.

11.4.4 The 'Dereferenced' Use of Simple NAME Identifiers.

Simple NAME identifiers are those which are not parts of structure templates.

If a simple NAME identifier appears in a HAL/S expression as if it were an ordinary identifier, then the value used in computing the expression is the value of the ordinary identifier pointed to by the NAME identifier. Similarly, if a simple NAME identifier appears on the left-hand side of an assignment, as if it were an ordinary identifier, then the value of the right-hand side is assigned to the ordinary identifier pointed to by the NAME identifier. These are examples of the 'dereferenced' use of NAME identifiers.

Whenever a NAME identifier appears in a HAL/S construct as if it were an ordinary identifier, the dereferencing process (to find the ordinary identifier pointed to) is implicitly being specified. Specifically, this still takes place when a subscripted NAME identifier appears as if it were an ordinary identifier.

Here the dereferencing takes place first, and then the subscripting is applied to the ordinary identifier pointed to:

examples of dereferenced NAME variables

```
DECLARE VECTOR(3), X, Y NAME;
DECLARE P NAME TASK;
Q: TASK;
    .
    .
    .
CLOSE;
    .
    .
```

if Y points to X, and P to Q then -

`TERMINATE P;`	Means terminate Q.
`Y = Y*Y;`	Puts the cross product of X with X in X.
$Y_1 = Y_3$;	Puts the third element of X into the first element.

Figure 11-18 NAME Variable Dereferencing Examples

A special construct to be described in Sections 11.4.5 and 11.4.6 is required to reference or change the value of a NAME identifier (as opposed to referencing or changing the value to which it points).

11.4.5 Referencing NAME Values.

The value of a NAME identifier is referenced or changed by using the NAME pseudo-function. This pseudo-function must also be used in order to gain access to the locations of ordinary HAL/S identifiers. The locations of values so indicated will be called NAME values. The necessity also arises for specifying null NAME values.

The following syntax diagram shows both the NAME pseudo-function construct as used for referencing NAME values, and the construct for specifying null NAME values.

SYNTAX:

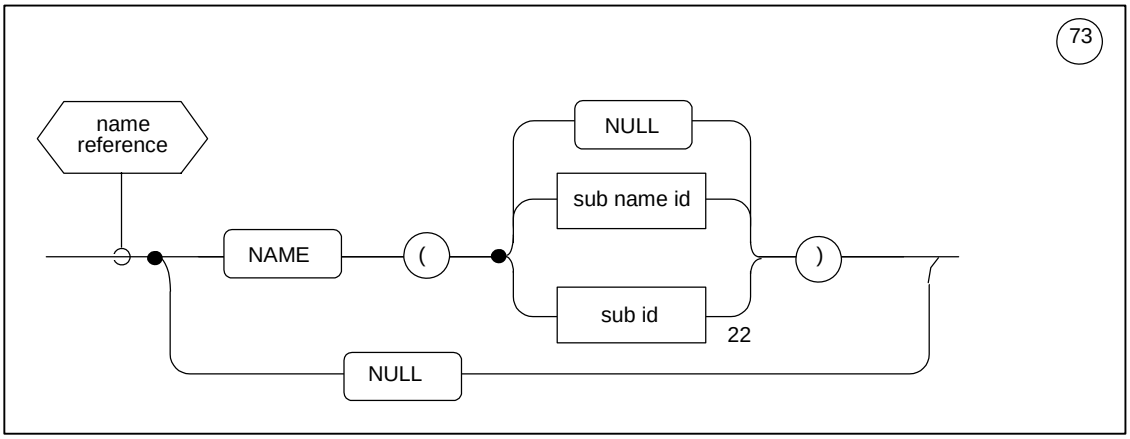


Figure 11-19 NAME reference - #73

SEMANTIC RULES:

1. <sub id> is any ordinary identifier, except an input parameter, a minor structure, an identifier declared with CONSTANT initialization, or an ACCESS controlled identifier to which assignment access is "denied" or not asked for. <sub name id> is any NAME identifier.
2. Either of the above forms may possibly be modified by subscripting legal for its type and organization. Note, however, the following specific exceptions:
   - No component subscripting is allowed for bit and character types.
   - If component subscripting is present, <sub id> or <sub name id> must be subscripted so as to yield a single (unarrayed) element.
   - If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

Example:

```
DECLARE V NAME ARRAY(3) VECTOR;
.
.
.
NAME(V*:1)
```

is illegal since it violates the second exception of semantic rule 2 above.

Figure 11-20 NAME Variable Subscripting Example

3. Any <sub id> must have the ALIGNED attribute.
4. NAME <identifier>s may not be declared with the ACCESS attribute (see Section 11.4.1, rule 2). This does not, however, imply that the NAME facility is independent of the ACCESS control: NAME references to <sub id>s with ACCESS control will compile without error only if implementation dependent ACCESS requirements for <sub id> are satisfied.
5. If <sub id> is unsubscripted, the construct delivers the location of the ordinary identifier specified. If it is subscripted, the construct delivers the location of the part of the specified identifier as determined by the form of the subscript. Subscripting can change the type and dimensions of <sub id> for matching purposes.
6. If <sub name id> is unsubscripted, the construct delivers the value of the NAME identifier specified. If it is subscripted, the value of the NAME identifier is taken to be the location of an ordinary identifier of compatible attributes, and the subscripting accordingly modifies the location delivered by the construct.
7. The two equivalent forms NULL and NAME(NULL) specify null NAME values.

Examples:

```
DECLARE X SCALAR,
          V VECTOR(3),
          NX NAME SCALAR,
          NV NAME VECTOR(3);
.
.
.
```

NAME(X)	yields the location of X.
NAME(NX)	yields the value of NX (i.e. the location pointed to by NX).
NAME(V$_2$)	yields the location of the second element of V.
NAME(NV$_3$)	yields the location of the third element of the vector pointed to by NV.

Figure 11-21 NAME Variable Referencing Examples

11.4.6 Changing NAME Values.

The value of NAME identifier is changed by using the NAME pseudo-function in an assignment context. The following syntax diagram shows the NAME pseudo-function used for assigning NAME values.

SYNTAX:

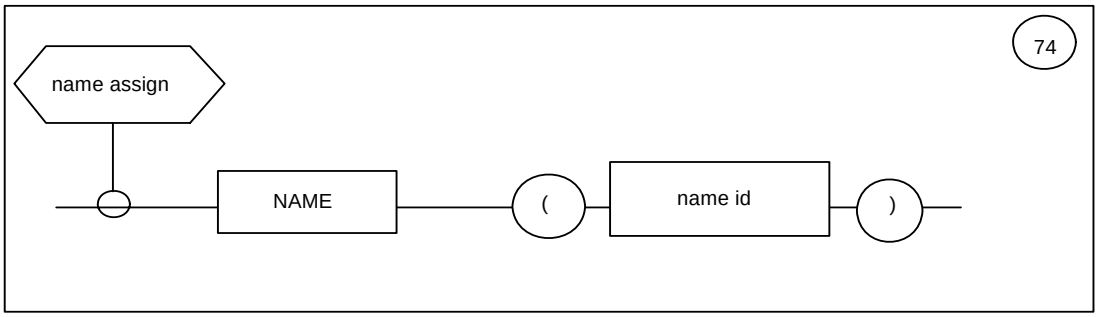


Figure 11-22 NAME assign #74

SEMANTIC RULE:

1. <name id> specifies any NAME identifier except an input parameter, whose NAME value is to be changed. <name id> may not be subscripted (except as noted in Section 11.4.11).

example:

```
DECLARE X NAME MATRIX;
```

NAME(X) in assignment context specifies that a new value is to be given to X.

Figure 11-23 NAME Assignment Example

11.4.7 NAME Assignment Statements.

The NAME assignment statement is the construct by which NAME values are assigned into NAME identifiers.

SYNTAX:

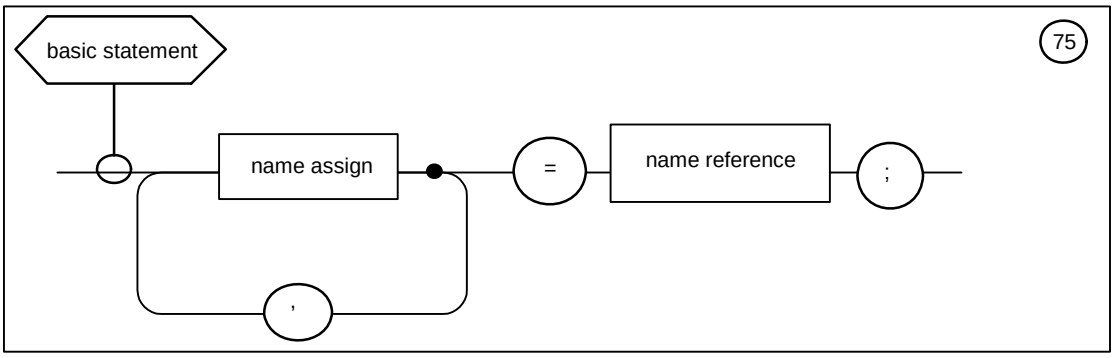


Figure 11-24 NAME assignment statement #75

SEMANTIC RULES:

1. The <name reference> and <name assign>s must possess arguments whose attributes are compatible in the sense described in Section 11.4.1.

11.4.8 NAME Value Comparisons.

The values of two <name reference>s may be compared to one another.

SYNTAX:

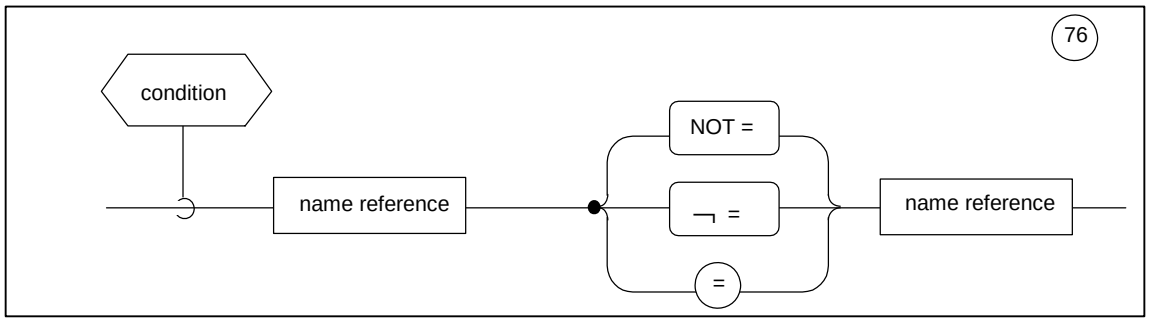


Figure 11-25 NAME conditional expression- #76

SEMANTIC RULES:

1. This <comparison> may be used in any syntax where other forms of <comparison> may be used; for example, in a <conditional operand> or as the <condition> controlling a DO WHILE.
2. Both <name reference>s must possess argument whose <attributes> are compatible in the sense described in Section 11.4.1 (with the exception of the REMOTE keyword which need not match).

examples:

```
DECLARE X SCALAR,
        NX NAME SCALAR,
    .
    .
    .
    .
NAME(NX)=NAME(X);      value of NX is location of X (NX points to X).
    .
    .
    .
IF NAME(NX)=NULL THEN RETURN;
          if NX contains a null value (points at no location) then return.
```

Figure 11-26 NAME Conditional Example

11.4.9 Argument Passage Considerations.

NAME values may be passed into procedure and functions provided that the corresponding formal parameters of the blocks in question have the NAME attribute. The following two syntax diagrams are systems language extensions of the earlier <normal function> and <call statement> syntax diagrams.

SYNTAX:

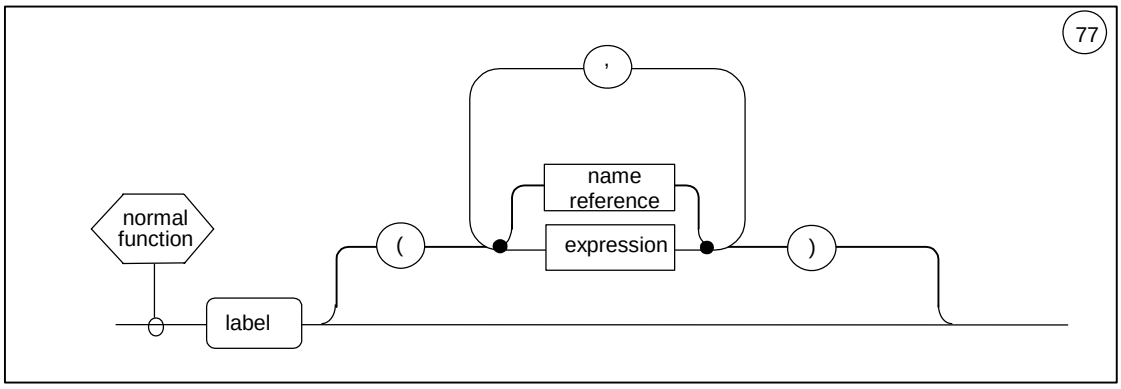


Figure 11-27 normal FUNCTION reference - #77

SYNTAX:

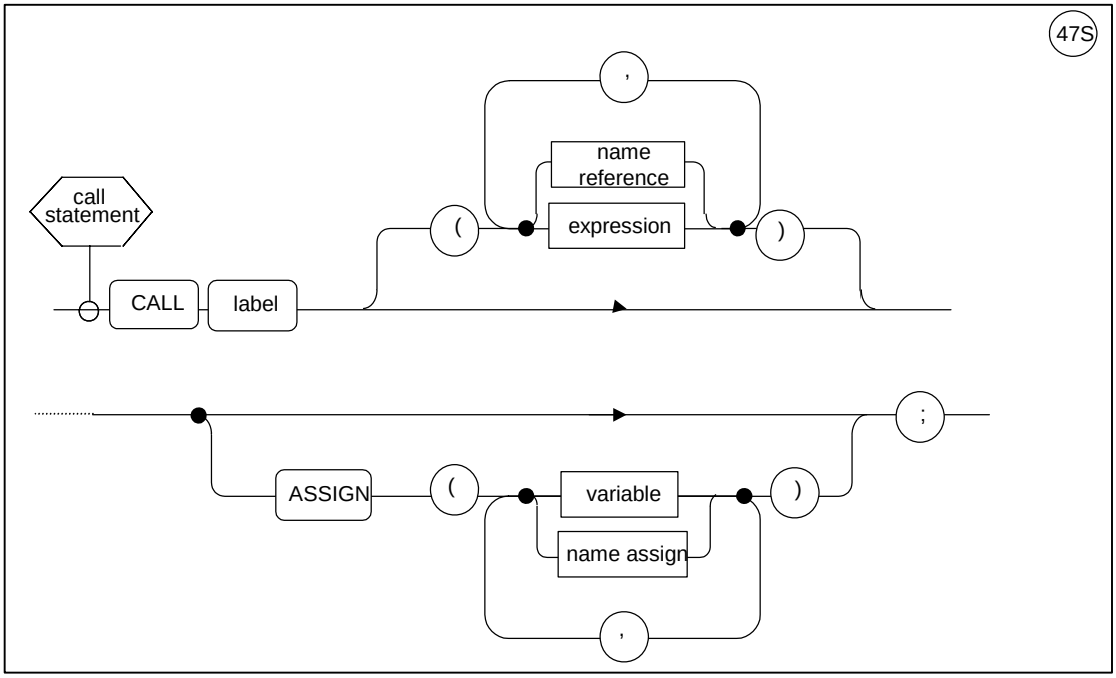


Figure 11-28 CALL STATEMENT with NAME - #47s

SEMANTIC RULES:

1. The formal parameters corresponding to <name reference> or <name assign> arguments of these block invocations must possess the NAME attribute.
2. The attributes of <name reference> and <name assign> arguments supplied in the <normal function> reference or <call statement> must be compatible with those of the formal parameters in the same sense as described in Section 11.4.1. The REMOTE attribute must match for NAME ASSIGN parameters.
3. If the argument(s) of that procedure or function invocation is not a <name reference> then the corresponding formal parameter must not have the NAME attribute.

Examples:

```
DECLARE XI SCALAR,
        X2 NAME SCALAR,
.
.
.
P: PROCEDURE(A,B) ASSIGN(C,D);
    DECLARE SCALAR,  A NAME,
                     B,
                     C NAME,
                     D;
    NAME(C) = NAME(A);
    NAME(C) = NAME(B); illegal - B is an input parame-
    ter
    .
    .
    .
CLOSE;
.
.
.
NAME(X2) = NAME(XI);
CALL P(NAME(XI),XI) ASSIGN(NAME(X2),XI);
```

Figure 11-29 NAME Variables as Parameters Example

11.4.10 Initialization.

NAME identifiers may be declared with initialization to point to some particular identifier. The form of NAME initialization is as follows:

SYNTAX:

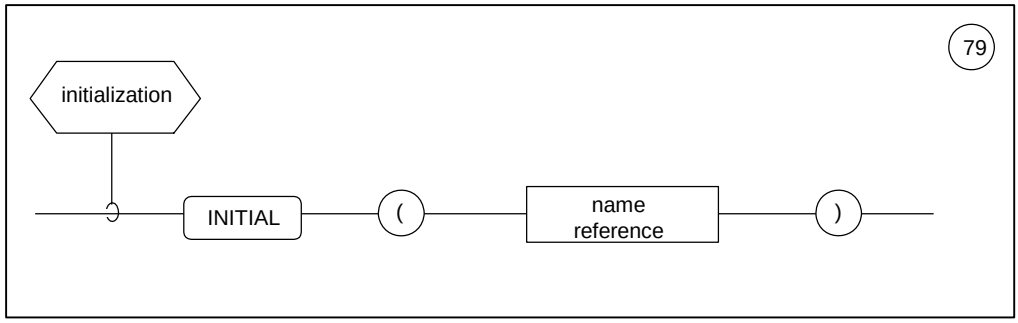


Figure 11-30 NAME initialization attribute #79

SEMANTIC RULES:

1. The argument of the <name reference> must be a previously declared <sub name id> or <sub id> with <attributes> compatible with the NAME identifier being declared in the same sense as described in Section 11.4.
2. Uninitialized NAME identifiers will have a NULL NAME value until the first NAME assignment.
3. The argument of a <name reference> may not itself possess the NAME attribute.

11.4.11 Notes on NAME Data and Structures.

The previous sections have introduced the various syntactical forms and uses of the NAME attribute, <name assign>s, and <name reference>s. The use of these NAME facilities with structure data merits further explanation since the implications of the various legal combinations are not always immediately apparent. Therefore, the purpose of this section is to continue further discussion of various aspects of NAME and structure usage by providing several examples.

STRUCTURE TERMINAL REFERENCES

Consider the structure template and structure data declarations below:

```
STRUCTURE A:
  1 C SCALAR,
  1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE, Z1, Z2, Z3;
```

Z1.B is a NAME identifier of A-structure type: its NAME value may be set to point to Z2 by the assignment

```
NAME(Z1.B) = NAME(Z2);
```

If this is done then it is legal to specify Z1.B.C as a qualified structure terminal name. If the appearance of B in the qualified name causes an implicit dereferencing process to occur such that if Z1.B.C is used in a dereferencing context, the ordinary structure terminal actually referenced is Z2.C. If the NAME value of Z1.B is changed by

```
NAME(Z1.B) = NAME(Z3);
```

then the appearance of Z1.B.C in a dereferencing context causes Z3.C to be referenced.

Pictorially:

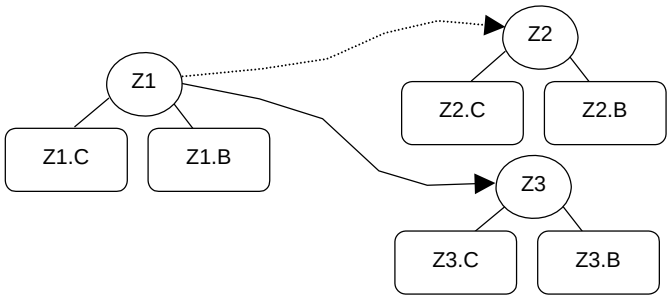


Figure 11-31 Structure NAME Dereference

Now Z1.B.B is itself in turn a NAME identifier of A-structure type, so that if the NAME assignment

```
NAME(Z1.B.B) = NAME(Z2);
```

is executed, then Z2.C may be referenced by using the qualified name Z1.B.B.C in a dereferencing context.

Pictorially:

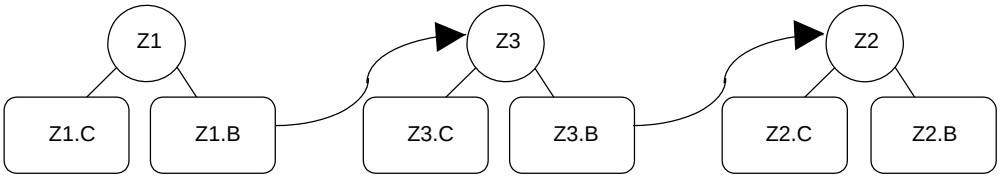


Figure 11-32 Structure NAME Dereference Chain

Clearly this implicit dereferencing in qualified name can extend chains of reference indefinitely. A particular consequence is the creation of a closed circular chain. If the following NAME assignment statements:

```
NAME(Z1.B)   = NAME(Z2);
NAME(Z1.B.B) = NAME(Z1);
```

are executed, then pictorially, the following closed loop is set up:

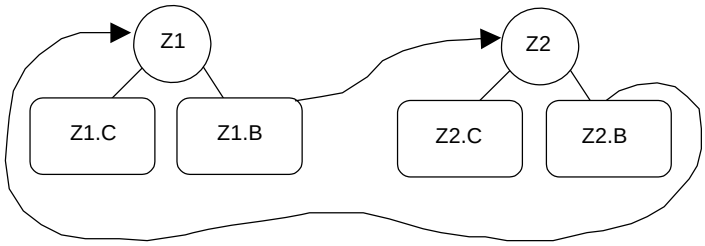


Figure 11-33 Structure NAME Dereference Loop

Care must clearly be taken when using this implicit multiple dereferencing, so that all links in the chain have previously been set up.

IMPLICATIONS OF SUBSCRIPTING STRUCTURE TERMINALS

Using the same A-structure template as before, the following declarations are legal:

```
DECLARE A-STRUCTURE(3), Y1, Y2, Y3, Y4;
```

One or more copies of Y1.C may be referred to by subscripting, for example:

```
Y1.C(2 AT 2;  (optional semicolon for clarity)
```

Note that Y1.B is a NAME identifier of A-structure type with 3 copies. One or more copies of it may therefore be assigned a NAME-value at one time. For example:

```
NAME(Y1.B(2 AT 2)) = NAME(Y2(2 AT 1));
```

In this assignment, the left hand side has arrayness: two copies of the Y1 structure. As a

result, two values will be defined by the statement. However, the right hand side has no arrayness, because the object pointed to is $Y2_{2\ AT\ 1}$. This is a two copy section of the structure Y2, with a unique starting location.

Pictorially:

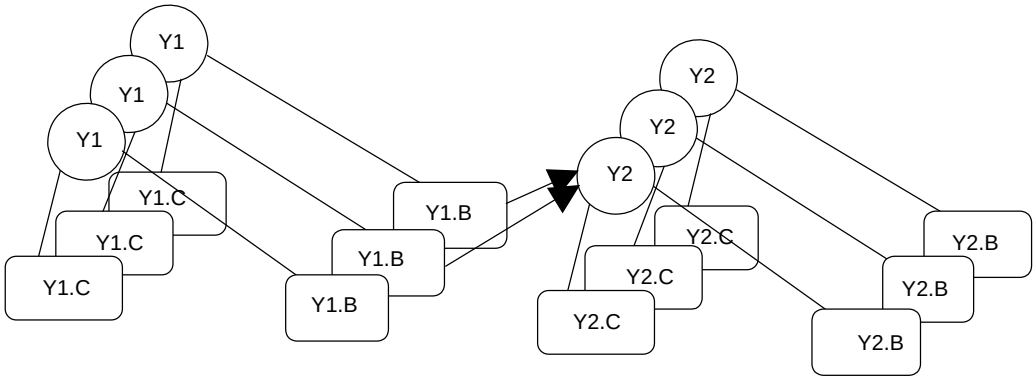


Figure 11-34 Structure NAME Dereference with Subscripting

Notice that in the above NAME assignment, a subscripted <name id> appears as an argument of the left-hand side NAME pseudo-function. Subscripts so appearing are legal only if they have the interpretation exemplified. The subscripting employed must also be unarrayed, as was mentioned earlier.

Further indirection may then be set up; thus, for example:

```
NAME(Y1.B.B2) = NAME(Y31);
```

Here, the subscript 2 on the left-hand argument refers to copies of Y1 (this can be its only interpretation). Hence, by virtue of the fact that $Y1.B_2$ has previously been set up to point to $Y2_1$, this assignment causes $Y2.B_1$ to point to $Y3_1$.

Arrayness will appear on both sides of a NAME assignment statement only when the assigned reference terminals of both sides possess the NAME attribute within structure variables with copies.

Consider the template:

```
STRUCTURE AA:
  1 C NAME SCALAR,
  1 D NAME VECTOR;
```

And the declaration:

```
DECLARE AA-STRUCTURE(3), YY1, YY2;
```

If the terminal element YY2.D is assigned to the terminal element YY1.D, the NAME assignment is arrayed since both sides contain three copies.

Thus:

```
NAME(YY1.D) = NAME(YY2.D);
```

causes the NAME values of YY2.D found in the three copies of YY2 to be transferred to the corresponding name variables in YY1.D. All the usual rules governing arrayed assignments apply is this case.

MANIPULATING STRUCTURES CONTAINING NAME TERMINALS

Since the NAME attribute may be applied to structure terminals, a definition of operations performed on such NAME terminals in ordinary structure assignments, comparisons, and I/O operations is required. The following general rules are applicable:

- For assignment statements and comparisons involving structure data with NAME terminals, operations are performed on NAME values without any dereferencing.

Examples:

```
STRUCTURE IOTA:
    1 LAMBDA NAME VECTOR,
    1 KAPPA SCALAR;
DECLARE ALPHA IOTA-STRUCTURE(10);
DECLARE BETA IOTA-STRUCTURE;
    .
    .
    .
ALPHA4 = BETA;
```

As a part of this assignment, the vector identifier (or NULL) pointed to by BETA.LAMBDA becomes the vector identifier pointed to by ALPHA.LAMBDA$_4$ as if a <name assignment statement> had been used.

```
IF ALPHA5 = BETA THEN CALL QUE_UPDATE;
```

In this IF statement, the structure comparison between the two variables (ALPHA$_5$ and BETA) is performed terminal by terminal as usual. For the NAME terminal LAMBDA of each structure operand, the effect is the same as if a <name comparison> had been used: Equality for the corresponding NAME terminals exists if they both point to the same ordinary identifier.

Figure 11-35 Structure NAME Manipulation

- For sequential I/O operations, all NAME terminals are totally ignored. NAME terminals can take part in FILE I/O.

examples:

```
STRUCTURE OMICRON:
  1 ALPHA SCALAR,
  1 BETA ARRAY(25) INTEGER SINGLE,
  1 GAMMA NAME MATRIX(10,10);
STRUCTURE TAU:
  1 ALPHA SCALAR,
  1 BETA ARRAY(25) INTEGER SINGLE;
DECLARE X OMICRON-STRUCTURE;
DECLARE Y TAU-STRUCTURE;
     .
     .
     .
READ(5) X;
```

The structure variable X is an OMICRON-STRUCTURE, whose template includes the NAME of a 10 x 10 matrix (GAMMA). Only the ordinary terminals are transferred from Channel 5 by this READ operation --- the value of X.ALPHA and the 25 values required for X.BETA. The NAME terminal X.GAMMA is ignored.

```
READ(5) Y;
```

The structure variable Y is a TAU-STRUCTURE, whose template omits the NAME terminal GAMMA found in the OMICRON-STRUCTURE, but is otherwise identical. The effect of this READ statement is the same as the previous statement as far as Channel 5 is concerned --- one value is read for Y.ALPHA and 25 values are read for Y.BETA.

Figure 11-36 Structure NAMEs with READ Statements

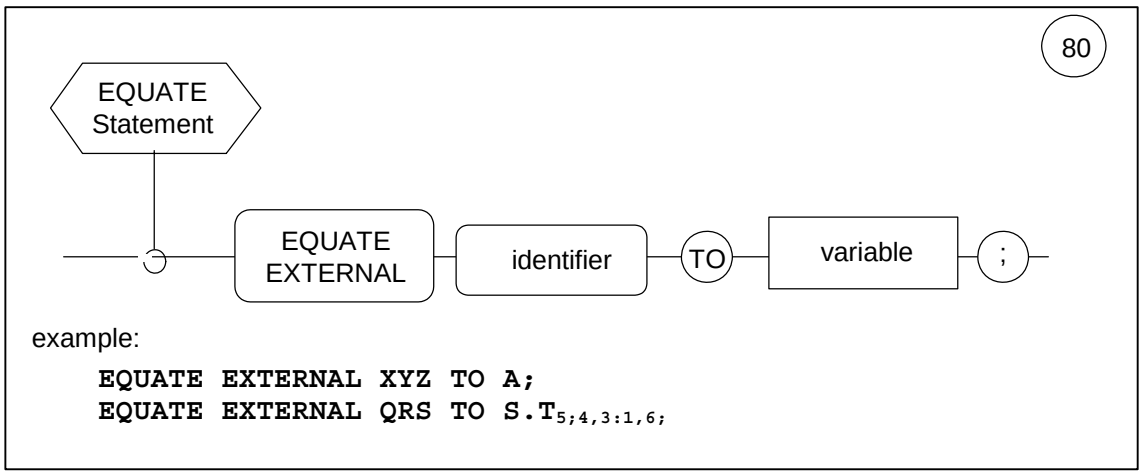
11.5 The EQUATE Facility.

This section describes the HAL/S EQUATE facility which allows a systems programmer to assign an external name to an element of a HAL/S data area.

Reference to HAL/S data items by HAL/S code is achieved by use of HAL/S identifiers. When such references occur across compilation unit boundaries, the Block Template provides the information necessary to generate the reference properly. If, however, the unit making reference to a HAL/S data item is not a HAL/S code block, the Block Template facility is unavailable. It is under these latter circumstances that the HAL/S EQUATE facility may be used to make the location of a HAL/S data item available to an external, non-HAL/S code block.

11.5.1 The EQUATE Statement.

SYNTAX:



example:

```
EQUATE EXTERNAL XYZ TO A;
EQUATE EXTERNAL QRS TO S.T5;4,3:1,6;
```

Figure 11-37 EQUATE Statement - #80

SEMANTIC RULES:

1. The EQUATE statement causes <identifier> to become an externally recognizable label of the HAL/S <variable>. The manner in which this is done is implementation dependent. The EQUATE statement has the effect of raising the name of <identifier> to a global external level such that it is known to whatever binders, loaders, link-editors, etc., are used by an implementation.
2. The number of characters of the <identifier> which participate in the external name created is implementation dependent.
3. The EQUATE statement does not constitute a HAL/S declaration. This implies that <identifier> may appear in a declaration statement and used in any manner consistent with that declaration. In the absence of such a declaration, <identifier> is not declared and may not be used anywhere else in the HAL/S code.
4. Duplication of <identifier>s among multiple EQUATE statements within a single compilation unit is subject to implementation dependent rules.
5. <variable> may be any HAL/S data item previously declared in the innermost scope containing the EQUATE statement.
6. If <variable> is subscripted, all subscripts must be computable at compile time.
7. The external name created by the EQUATE statement will be associated with the memory location of the first (or only) element specified by <variable>.
8. Attempts to associate external names with HAL/S data items which are not located integrally at addressable memory locations or discontiguous memory locations are subject to implementation restrictions.

11.5.2 EQUATE Statement Placement.

The following diagram is a system language extension of the Declare Group syntax diagram in Section 4.1. The modification shows how the EQUATE statement fits into the declaration structure of HAL/S.

SYNTAX:

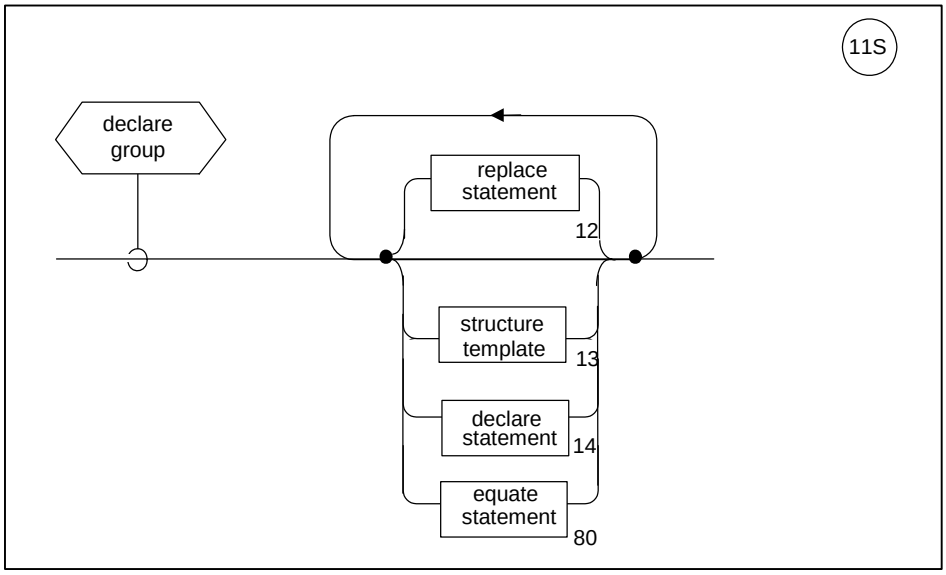


Figure 11-38 declare group - #11s

This page is intentionally left blank.

Appendix A - A SYNTAX DIAGRAM SUMMARIES

A.1 SYNTAX PRIMITIVE REFERENCES

The syntax diagrams in this Specification are numbered sequentially. The CONTENTS of the Specification state which diagrams are in each section.

The following table shows where the HAL/S syntactical primitives (excluding reserved words and special characters) are referred to.

NOTES:

1. Primitives are listed in alphabetical order.
2. Numbers enclosed in [ ] denote indirect references to the primitive. Explanations are given in the accompanying Semantic Rules.

Syntactical Primitive	Diagram Number	Page
<arith var name>	[19]	[5-2]
	20	5-3
<argument>	12.1	4-3
<bit literal>	19	5-2
	20	5-3
<char literal>	[18]	[4-18]
	29	6-8
<char var name>	[18]	[4-18]
	19	5-2
	20	5-3
<event var name>	19	5-2
	20	5-3
<identifier>	8	3-8
	9	3-9
	12	4-2
	13	4-7
	14	4-10
	14s	11-11
	13s	11-14
<label>	2	3-2
	3	3-3
	4	3-4
	5	3-5
	6	3-6
	10	3-11
	[18]	[4-18]
	38	6-18

Syntactical Primitive	Diagram Number	Page
	45	7-1
	46	7-2
	47	7-5
	48	7-7
	50	7-9
	51	7-9
	52	7-10
	53	7-11
	54	7-12
	55	7-13
	56	7-14
	57	8-3
	58	8-5
	59	8-6
	60	8-7
	61	8-8
	62	8-9
	63	9-2
	64	9-4
	65	10-2
	66	10-4
	68	10-19
	53s	11-9
	54s	11-10
	77	11-20
	47s	11-21
<number>	13	4-7
	15	4-11
	16	4-14
	[18]	[4-18]
	25	6-5
	63	9-2
	64	9-4
	68	10-19
	83	10-8
	85	10-11
	86	10-12
	87	10-14
	88	10-15

Syntactical Primitive	Diagram Number	Page
<process-event name>	13s 27	11-14 6-7
<template name>	37 17	6-17 4-15
<text>	12 [12.1]	4-2 [4-3]

A.2 SYNTAX DIAGRAM CROSS REFERENCES

The following table shows where non-primitive syntactical terms are defined and referenced.

NOTES:

1. Terms are listed in alphabetical order.
2. <radix> is included even though it has no syntactical diagram, because, for the purposes of the Specification, it was not regarded as a primitive. Its definition is included in the Semantic Rules accompanying the syntax diagrams where it is referred to.
3. Note that an “s” suffix identifies a modified Systems Language Diagram.

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<arith conversion>	39	6	6-20	25, 25s
<arith exp>	24	6	6-2	15, 17, 18, 22, 23, 25, 28, 32, 39, 51, 53, 54, 57, 60, 61, 67, 68, 25s, 54s
<arith inline>	25s	11	11-4	25
<arith %macro>	25s	11	11-4	25
<arith operand>	25	6	6-5	24, 25s
<arith var>	19	5	5-2	20, 25, 53, 54, 25s, 54s
<array sub>	22	5	5-6	21
<attributes>:				
data	15	4	4-11	
label	16	4	4-14	16s
name	16s	11	11-13	44, 45
<basic statement>:				
assignment	46	7	7-2	
name	75	11	11-19	47s
CALL	47	7	7-5	47s
name	47s	11	11-21	
CANCEL	58	8	8-5	
DO…END	49	7	7-8	
EXIT	56	7	7-14	
FILE	68	10	10-19	
GO TO	56	7	7-14	
name assign	74	11	11-18	
null	56	7	7-14	75, 76, 77, 47s
ON ERROR	63	9	9-2	
READ	65	10	10-2	
READALL	65	10	10-2	
REPEAT	56	7	7-14	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
RESET	62	8	8-9	
RETURN	48	7	7-7	
SCHEDULE	57	8	8-3	
SEND ERROR	64	9	9-4	
SET	62	8	8-9	
SIGNAL	62	8	8-9	
TERMINATE	59	8	8-6	
UPDATE PRIORITY	61	8	8-8	
WAIT	60	8	8-7	
WRITE	66	10	10-4	
<bit conversion>	40	6	6-23	27, 27s
<bit exp>	26	6	6-6	23, 27, 33, 41, 45, 52, 53, 54, 27s, 54s
<bit inline>	27s	11	11-5	27
<bit %macro>	27s	11	11-5	27
<bit operand>	27	6	6-7	26, 27s
<bit pseudo-var>	42	6	6-26	20, 27, 27s
<bit var>	19	5	5-2	20, 27, 27s
<char conversion>	41	6	6-25	29, 29s
<char exp>	28	6	6-8	23, 29, 34, 30, 29s
<char inline>	29s	11	11-6	29
<char %macro>	29s	11	11-6	29
<char operand>	29	6	6-8	28, 29s
<char var>	19	5	5-2	20, 29, 29s
<closing>	10	3	3-11	2, 3, 4, 5, 6, 69
<comparison>:				31
arithmetic	32	6	6-12	
bit	33	6	6-13	
character	34	6	6-14	
structure	35	6	6-15	
<compilation>	1	3	3-1	
<component sub>	22	5	5-6	6-21
<compool block>	5	3	3-5	1
<compool header>	7	3	3-7	5, 6
<compool template>	6	3	3-6	1
<condition>	30	6	6-10	31, 45, 52, 53, 54, 54s
name	76	11	11-19	
<conditional operand>	31	6	6-11	30
<declare group>	11	4	4-2	2, 3, 4, 5, 6, 69
<declare statement>	14	4	4-10	11, 11s, 14s
name	14s	11	11-11	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<do statement>:				
END	49	7	7-8	49, 49s
CASE	51	7	7-9	
discrete FOR	53	7	7-11	
temporary var	53s	11	11-9	
iterative FOR	54	7	7-12	
temporary var	54s	11	11-10	
simple	50	7	7-9	
UNTIL	52	7	7-10	
WHILE	52	7	7-10	
<end statement>	55	7	7-13	49, 49s
<equate statement>	80	11	11-28	11s
<event exp>	36	6	6-16	37, 57, 60
<event operand>	37	6	6-17	36
<event var>	19	5	5-2	20, 27, 37, 62
<expression>	23	6	6-1	18, 38, 39, 40, 41, 42, 46, 47, 48, 66, 68, 70
<file exp>	68	10	10-19	68
<function block>	3	3	3-3	1, 2, 3, 4, 49, 49s
<function header>	9	3	3-9	3, 6
<function template>	6	3	3-6	1
<inline function>	69	11	11-2	
<initial list>	18	4	4-18	18
<initialization>	18	4	4-18	15
name	79	11	11-22	16s
<i/o control>	67	10	10-6	65, 66
<%macro>	70	11	11-3	25s
typeless %macro	71	11	11-7	
<name>	14s	11	11-11	
<name assign>	75	11	11-19	
<normal function>	38	6	6-18	25, 27, 29, 77
name	77	11	11-20	
<precision>	43	6	6-28	1, 2, 3, 4, 49, 49s
<procedure block>	3	3	3-3	3, 6
<procedure header>	8	3	3-8	3, 6
<procedure template>	6	3	3-6	1
<program block>	2	3	3-2	1
<program header>	7	3	3-7	2
<radix>	Note 2.	6		40, 41
<replace statement>	12	4	4-2	11, 11s
parametric	12.1	4	4-3	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<statement>:				
basic	44	7	7-1	
IF	45	7	7-1	
temporary	72	11	11-8	
<struct inline>	29.1s	11	11-6	29.1
<struct %macro>	29.1s	11	11-6	29.1
<structure exp>	29.1	6	6-9	
<structure sub>	22	5	5-6	21
<structure template>	13	4	4-7	11, 11s
name	13s	11	11-14	
<structure var>	19	5	5-2	20, 23, 35, 29, 15, 29.1
<sub exp>	22	5	5-6	22
<sub nameid>	73	11	11-16	
<subscript>	21	5	5-4	19, 39, 40, 41, 42
<task block>	3	3	3-3	2, 49, 49s
<task header>	7	3	3-7	3
<temporary statement>	49s	11	11-8	53s, 54s
	72	11	11-8	
<type spec>	17	4	4-15	9, 15, 16, 69
<update block>	4	3	3-4	2, 3, 49, 69, 49s
<update header>	7	3	3-7	4
<variable>	20	5	5-3	42, 46, 47, 65, 68

A.3 SYNTAX DIAGRAM LISTING

Appendix B - HAL/S KEYWORDS

The following table of keywords excludes built-in functions and %macro names.

ACCESS	ELSE	NAME	SKIP
AFTER	END	NONHAL	STATIC
ALIGNED	EQUATE	NOT	STRUCTURE
AND	ERROR	NULL	SUBBIT
ARRAY	EVENT		SYSTEM
ASSIGN	EVERY	OCT	
AT	EXCLUSIVE	OFF	TAB
AUTOMATIC	EXIT	ON	TASK
	EXTERNAL	OR	TEMPORARY
BIN			TERMINATE
BIT	FALSE	PAGE	THEN
BOOLEAN	FILE	PRIORITY	TO
BY	FOR	PROCEDURE	TRUE
	FUNCTION	PROGRAM	
CALL			UNTIL
CANCEL	GO	READ	UPDATE
CASE		READALL	
CAT	HEX	REENTRANT	VECTOR
CHAR		REMOTE	
CHARACTER	IF	REPEAT	WAIT
CLOSE	IGNORE	REPLACE	WHILE
COLUMN	IN	RESET	WRITE
COMPOOL	INITIAL	RETURN	
CONSTANT	INTEGER	RIGID	
DEC	LATCHED	SCALAR	
DECLARE	LINE	SCHEDULE	
DENSE	LOCK	SEND	
DEPENDENT		SET	
DO	MATRIX	SIGNAL	
DOUBLE		SINGLE	

This page is intentionally left blank.

Appendix C - BUILT-IN FUNCTIONS

HAL/S typically supports the following set of built-in functions. Minor variations may arise between implementations.

ARITHMETIC FUNCTIONS	
arguments may be INTEGER or SCALAR types in functions with one argument, result type matches argument type (except as specifically noted) in functions with two arguments, unless specifically specified, result type is scalar if either or both arguments are scalar; otherwise the result type is integer arrayed arguments cause multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match	
Name, Arguments	**Comments**
ABS(α)	$\lvert \alpha \rvert$
CEILING(α)	smallest integer $\geq \alpha$
DIV(α,β)	integer division α/β (arguments rounded to integers)
FLOOR (α)	largest integer $\leq \alpha$
MIDVAL(α,β,γ)	the value of the argument which is algebraically between the other two. If two or more arguments are equal, the multiple value is returned. Result is always scalar.
MOD(α,β)	α MOD β
ODD(α)	TRUE 1 if α odd FALSE 0 if α even — result is BOOLEAN
REMAINDER(α,β)	signed remainder of integer division α/β (argument rounded to integer)
ROUND(α)	nearest integer to α
SIGN(α)	+1 $\alpha \geq 0$ -1 $\alpha < 0$
SIGNUM(α)	+1 $\alpha > 0$ 0 $\alpha = 0$ -1 $\alpha < 0$
TRUNCATE(α)	largest integer $\leq \lvert \alpha \rvert$ times SIGNUM (integer (α))

ALGEBRAIC FUNCTIONS	
• arguments may be integer or scalar types -- conversion to scalar occurs with integer arguments • result type is scalar • arrayed arguments cause multiple invocations of the function, one for each array element • angular values are supplied or delivered in radians	
Name, Arguments	**Comments**
ARCCOS(α)	$\cos^{-1}\alpha$, $\lvert\alpha\rvert \leq 1$
ARCCOSH(α)	$\cosh^{-1}\alpha$, ≥ 1
ARCSIN(α)	$\sin^{-1}\alpha$, $\lvert\alpha\rvert \leq 1$
ARCSINH(α)	$\sinh^{-1}\alpha$
ARCTAN2(α,β)	$-\pi < \tan^{-1}(\alpha/\beta) \leq \pi$ Proper Quadrant if: $\left.\begin{array}{l}\alpha = k \sin\theta \\ \beta = k \cos\theta\end{array}\right\} k > 0$
ARCTAN(α)	$\tan^{-1}\alpha$
ARCTANH(α)	$\tanh^{-1}\alpha$, $\lvert\alpha\rvert < 1$
COS(α)	$\cos\alpha$
COSH(α)	$\cosh\alpha$
EXP(α)	e^{α}
LOG(α)	$\log_e\alpha$, $\alpha > 0$
SIN(α)	$\sin\alpha$
SINH(α)	$\sinh\alpha$
SQRT(α)	$\sqrt{\alpha}$, $\alpha \geq 0$
TAN(α)	$\tan\alpha$
TANH(α)	$\tanh\alpha$

VECTOR-MATRIX FUNCTIONS	
• arguments are vector or matrix types as indicated • result types are as implied by mathematical operation • arrayed arguments cause multiple invocation of the function, one for each array element	
Name, Arguments	**Comments**
ABVAL(α)	length of vector α.
DET(α)	determinant of square matrix α.
INVERSE(α)	inverse of a nonsingular square matrix α.
TRACE(α)	sum of diagonal elements of square matrix α.
TRANSPOSE(α)	transpose of matrix α.
UNIT(α)	unit vector in same direction as vector α.

MISCELLANEOUS FUNCTIONS		
• arguments are as indicated; if none are indicated the function has no arguments • result type is as indicated		
Name, Arguments	**Result Type**	**Comments**
CLOCKTIME	scalar	returns time of day
DATE	integer	returns date (implementation dependent format)
ERRGRP	integer	returns group number of last error detected, or zero
ERRNUM	integer	returns number of last error detected, or zero
PRIO	integer	returns priority of process calling function
RANDOM	scalar	returns random number from rectangular distribution over range 0-1
RANDOMG	scalar	scalar returns random number from Gaussian distribution mean zero, variance one.
RUNTIME	scalar	scalar returns Real Time Executive clock time (Section 8.)
NEXTIME (<label>)	scalar	<label> is the name of a program or task. The value returned is determined as follows: a) If the specified process was scheduled with the REPEAT ENTRY option and has begun at least one cycle of execution, then the value is the time the next cycle will begin. b) If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution. c) Otherwise, the value is equal to the current time (RUNTIME function).

MISCELLANEOUS FUNCTIONS (Cont'd)		
SHL(α,β)	Same as α	α may be integer or scalar type. β may be integer or scalar type. If α is integer type, the result is an integer whose internal binary representation is that of α shifted left by β bit locations. The signed nature of the integer α is taken into account in an implementation dependent manner which depends upon the number system and word size of the target computer. Scalar types will be converted to integer types prior to shifting. If β is a literal or constant, then it is illegal for it to be outside the range (1,63). Otherwise, only the 6 right-most bits of the value are used, restricting the range to (0,63). Arrayed arguments produce multiple invocations of the function, one for each array element-arrayness of arrayed arguments must match.
SHR(α,β)	Same as α	α may be integer or scalar type. β must be integer type. Results are as defined for the SHL function except that all shifting occurs to the right. The SHR is an arithmetic shift (sign bit is propagated). Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.

CHARACTER FUNCTIONS

- first argument is character type - second argument is as indicated (any argument indicated as character type may also be integer or scalar, whereupon conversion to character type is implicitly assumed)
- result type is as indicated
- arrayed arguments produce multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match

Name, Arguments	Result Type	Comments
INDEX(α,β)	integer	β is character type - if string β appears in string α , index pointing to the first character of β is returned; otherwise zero is returned
LENGTH(α)	integer	returns length of character string
LJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the right with blanks. β ≥ length (α)
RJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the left with blanks. β ≥ length (α)
TRIM(α)	character	leading and trailing blanks are stripped from α

BIT FUNCTIONS

- arguments are bit type
- result is bit type
- arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match

Name, Arguments	Result Type	Comments
XOR(α,β)	bit	Result is Exclusive OR of α and β. Length of result is length of longer argument. Shorter argument is left padded with binary zeros to length of longer argument.

ARRAY FUNCTIONS

- arguments are n-dimensional arrays where n is arbitrary.
- asterisk size REMOTE ARRAYs are restricted for the MAX, MIN, PROD and SUM functions unless a finite size array is also part of the argument
- arguments are integer or scalar type
- result type matches argument type and is unarrayed

Name, Parameters	Comments
MAX(α)	maximum of all elements of α.
MIN(α)	minimum of all elements of α.
PROD(α)	product of all elements of α.
SUM(α)	sum of all elements of α.

SIZE FUNCTION

Name, Argument	Comments
SIZE(α)	One of the following must hold: • α is an unsubscripted arrayed variable with a one-dimensional array specification - function returns length of array. • α is an unsubscripted major structure with a multiple copy specification - function returns number of copies. • α is an unsubscripted structure terminal with a one-dimensional array specification - function returns length of array. Result is of integer type

Appendix D - STANDARD CONVERSION FORMATS

In relatively limited circumstances HAL/S allows conversion between scalar, integer, bit, and character types. The following rules govern such conversions.

CONVERSIONS TO INTEGER TYPE:

- A bit type is converted to integer type by regarding it as the bit pattern of a signed integer of the desired precision (halfword or fullword). Left padding with binary zeros, or left truncation may occur.
- A scalar type is converted to integer type by rounding to the nearest whole number. Overflow errors may occur if the absolute value of the scalar type is too large to be represented as an integer of the desired precision.
- A character type is convertible to integer type only if its value represents a signed whole number (e.g., '-604') otherwise an error condition occurs. An error condition also occurs if the whole number is too large to be represented as an integer of the desired precision.

CONVERSIONS TO SCALAR TYPE:

- An integer type is converted directly to scalar form. The precision of the INTEGER is maintained (i.e. INTEGER SINGLE is converted to SCALAR SINGLE, INTEGER DOUBLE is converted to SCALAR DOUBLE). If a conversion from an INTEGER DOUBLE to a SCALAR SINGLE is forced through explicit conversion or assignment, some decimal places of accuracy may be lost during the conversion.
- A bit type is converted to scalar type by first converting it to an integer type with adequate precision to ensure no loss of accuracy and then applying the integer to scalar conversion rule above.
- A character type is convertible to scalar type only if its value represents a legal scalar or integer valued literal (e.g., '-1.5E-7'). See Section 2.3.3 for details of arithmetic literals. Other values cause error conditions to arise.

CONVERSIONS TO BIT TYPE:

- An integer type is converted to a bit string of maximum length (for INTEGER DOUBLE), or a halfword bit string (for INTEGER SINGLE) in an unsubscripted BIT conversion function by left padding with binary zeros as required. The value is the bit pattern of the integer.
- A scalar type is first converted to double precision integer type according to the rule already given, and the integer to bit conversion rules are then applied.
- A character type is convertible to bit type only if its value is a string of '1's and '0's, and blanks, (but not all blanks), otherwise an error condition arises. The result of the conversion is always a maximum length bit string, irrespective of the argument type. If the argument has more than N bits, where N is the maximum allowable length of a bit operand, then only the N right-most are used. If the argument has fewer than N bits, the string is padded on the left with binary zeros.

CONVERSION TO CHARACTER TYPE:

- An integer type is converted to the representation:

  dddd (positive)
  -dddd (negative)

  where dddd represents an arbitrary number of decimal digits. Leading zeros are suppressed yielding a variable length result.

- A scalar type is converted to the representation:

  ~~b~~d.ddddE±dd (positive)

  -d.ddddE±dd (negative)

  (except scalar 0 is converted to 0.0).

  The number of decimal digits d in the fractional part and exponent are implementation and precision dependent. The digit to the left of the decimal point is non-zero. There are no imbedded blanks. Leading zeros in the exponent are not suppressed. The representation includes a leading blank (~~b~~) if the scalar is positive. In all cases, the result is fixed in length.

- A bit type is converted to a character string of ‘1’s and ‘0’s corresponding to the binary representation of the bit string argument.

Appendix E - STANDARD EXTERNAL FORMATS

Corresponding to each data type there exists a "standard external format" for the representation of its values on sequential I/O files. In any implementation, the standard external format on output is fixed; on input the user has a certain flexibility in the format he can use.

OUTPUT FORMATS

1. Integer Type:
   - The value of an integer is represented by a string of decimal digits, preceded if it is negative by a "-" sign. Leading zeroes are suppressed.
   - The string of digits is right justified in a field of fixed width. The width depends upon the implementation, and on the precision of the integer.
2. Scalar Type:
   - If the value of a scalar is positive it is represented by:

     ~~b~~ d.ddddddd E±dd

     where d represents a decimal digit. One non-zero digit appears before the decimal point. The numbers of digits in the fractional part and exponent are fixed, and depend on the implementation and the precision of the scalar. Leading zeroes in the exponent are not suppressed. The representation includes a leading blank.
   - A negative value has the same form except that a "-" sign precedes the first decimal digit.
   - If the value is exactly zero, it is represented as 0.0.
   - The representation of a scalar is contained in a field of fixed width. The width is dependent on the implementation and the precision of the scalar. Justification is such that the decimal point occupies a fixed, precision dependent position in the field.
3. Bit Type (including BOOLEAN):
   - There are two different representations of values of bit variables.
   - The first representation consists of a string of binary digits corresponding to the bit variable. Leading binary zeroes are not suppressed. The field width is equal to the number of binary digits in the string plus an inserted blank following every fourth digit (to enhance readability). This form is not compatible with the READ input (see Section 10.1.1).
   - In the alternate representation, the string of binary digits plus inserted blanks is enclosed in apostrophes. The field width is equal to the total number of digits, blanks, and two apostrophes.
4. Character Type:
   - There are two different representations of values of character variables.
   - The first representation merely consists of the string of characters comprising the value. The field width is equal to the number of characters in the string. This representation is not compatible with READ input (see Section 10.1.1).

- In the alternate representation, the string of characters is enclosed in apostrophes, and all internal apostrophes are converted to apostrophe pairs. The field width is equal to the total number of characters in the string, including added apostrophes.

NOTE: The two alternate representations for bit and character type occur on paged and unpaged output, respectively.

INPUT FORMATS

1. Scalar and Integer Types:
   - There are three basic representations: whole-number, floating point, and fraction.
   - The whole-number representation consists of a string of decimal digits preceded by an optional "-" sign. The maximum number of digits allowed is implementation dependent. Conversion to mantissa-exponent form takes place for scalar types.
   - The floating-point representation is either

     ddd.dddd

     or

     dddd.dddd {E, B, H} ±dd

     where d is a decimal digit. Any number of digits is allowed in the mantissa to an implementation dependent maximum. The decimal point may appear in any position. E, B, and H represent the exponent digits to be powers of 10, 2, and 16, respectively. A choice of one is indicated. The maximum number of digits in the exponent is implementation dependent. For bit and integer types, the representation is rounded to the nearest integral value. For bit types, the binary representation of the result is taken.
   - The floating point representation may be prefixed by + or - signs to indicate the sign of the value. Without such prefix the value is positive.
2. Character Type:
   - The representation of character type is a string of characters from the HAL/S extended set enclosed in apostrophes. The number of characters may vary between zero (a "null string") and an implementation dependent maximum. Within the string apostrophes must be represented by an apostrophe pair.
3. Bit Type:
   - The representation of bit type is a string of '1's and '0's enclosed in apostrophes. Imbedded blanks are ignored. The number of digits may vary between one and an implementation dependent maximum.

Appendix F - COMPILE-TIME COMPUTATIONS

References are made in the text to expressions which must be computable at compile time. In particular, the following constructs make use of them:

- declaration of dimensions;
- initialization;
- subscripting.

Subsets of arithmetic, bit, and character expressions are guaranteed to be computable at compile time.

ARITHMETIC EXPRESSIONS (see Section 6.1.1)

1. <arith exp>s of integer and scalar type only can be computable at compile time.
2. The operators of such <arith exp>s are limited to:

   \+
   \-
   * (multiply)
   /
   **

3. The <arith operand>s of such <arith exp>s may either be <number>s or unarrayed, unsubscripted, simple variables (see Section 4.5) of integer or scalar type. Such variables must previously have been declared and initialized using the CONSTANT form (see Section 3.8).
4. The following built-in functions are also legal:

   ```
   SIN     EXP    DATE
   COS     LOG    CLOCKTIME
   TAN     SQRT
   ```

   DATE and CLOCKTIME are only computed at compile time if they appear in an <initialization> construct.

BIT EXPRESSIONS (see Section 6.1.2)

1. The operators which may appear in <bit exp>s computable at compile time are:

   ¬
   &
   |

   The <bit operand>s of such <bit exp>s must be either <bit literal>s or unarrayed, unsubscripted, simple variables of bit type. Such variables must previously have been declared and initialized using the CONSTANT form.

CHARACTER EXPRESSIONS (see Section 6.1.3)

1. The catenation operator (||) may only appear in <char exp>s computable at compile time.
2. The <char operand>s of such <char exp>s must be either <char literal>s, or unarrayed, unsubscripted, simple variables of character type. Such variables must previously have been declared and initialized using the CONSTANT form.
   In some implementations, additional forms may also be computed at compile time. They will not, however, be regarded as legal in contexts where compile time computability is enforced semantically.

Appendix G - BNF WORKING GRAMMAR of HAL/S

```
1     <COMPILATION> :: = <COMPILE LIST> _|_

2     <COMPILE LIST> :: = <BLOCK DEFINITION>
3                       | <COMPILE LIST> <BLOCK DEFINITION>

4     <ARITH EXP> :: = <TERM>
5                    | + <TERM>
6                    | -1 <TERM>
7                    |   <ARITH EXP> + <TERM>
8                    |   <ARITH EXP> -1 <TERM>

9     <TERM>  :: = <PRODUCT>
10               | <PRODUCT> / <TERM>

11     <PRODUCT> :: = <FACTOR>
12                  | <FACTOR> * <PRODUCT>
13                  | <FACTOR> . <PRODUCT>
14                  | <FACTOR> <PRODUCT>

15     <FACTOR> :: = <PRIMARY>
16                 | <PRIMARY> <**> <FACTOR>

17     <**> :: = **

18     <PRE PRIMARY>  :: = (<ARITH EXP>)
19                        | <NUMBER>
20                        | <COMPOUND NUMBER>

21     <ARITH FUNC HEAD> :: = <ARITH FUNC>
22                             | <ARITH CONV> <SUBSCRIPT>

23     <ARITH CONV> :: = INTEGER
24                      | SCALAR
25                      | VECTOR
26                      | MATRIX

27   <PRIMARY> :: = <ARITH VAR>

28   <PRE PRIMARY> :: = <ARITH FUNC HEAD> (<CALL LIST>)

29   <PRIMARY> :: = <MODIFIED ARITH FUNC>
30                | <ARITH INLINE DEF> <BLOCK BODY> <CLOSING>;
31                | <PRE PRIMARY>
32                | <PRE PRIMARY> <QUALIFIER>

33     <OTHER STATEMENT> :: = <ON PHRASE> <STATEMENT>
34                           | <IF STATEMENT>
35                           | <LABEL DEFINITION> <OTHER STATEMENT>

36     <STATEMENT> :: = <BASIC STATEMENT>
37                    | <OTHER STATEMENT>

38     <ANY STATEMENT> :: = <STATEMENT>
39                         | <BLOCK DEFINITION>

40     <BASIC STATEMENT> :: = <LABEL DEFINITION> <BASIC STATEMENT>
41                           | <ASSIGNMENT>;
42                           | EXIT;
43                           | EXIT <LABEL>;
44                           | REPEAT;
45                           | REPEAT <LABEL>;
46                           | GO TO <LABEL>;
47                           |;
48                           | <CALL KEY>;
49                           | <CALL KEY> (<CALL LIST>);
50                           | <CALL KEY> <ASSIGN> (<CALL ASSIGN LIST>);
51                           |<CALL KEY>(<CALL LIST>)<ASSIGN> (<CALL ASSIGN LIST>);
52                           | RETURN;
53                           | RETURN <EXPRESSION>;
```

```
54                          | <DO GROUP HEAD> <ENDING>;
55                          | <READ KEY>;
56                          | <READ PHRASE>;
57                          | <WRITE KEY>;
58                          | <WRITE PHRASE>;
59                          | <FILE EXP> = <EXPRESSION>;
60                          | <VARIABLE> = <FILE EXP>;
61                          | <WAIT KEY> FOR DEPENDENT;
62                          | <WAIT KEY> <ARITH EXP>;
63                          | <WAIT KEY> UNTIL <ARITH EXP>;
64                          | <WAIT KEY> FOR <BIT EXP>;
65                          | <TERMINATOR>;
66                          | <TERMINATOR> <TERMINATE LIST>;
67                          | UPDATE PRIORITY TO <ARITH EXP>;
68                          | UPDATE PRIORITY <LABEL VAR> TO <ARITH EXP>;
69                          | <SCHEDULE PHRASE>;
70                          | <SCHEDULE PHRASE> <SCHEDULE CONTROL>;
71                          | <SIGNAL CLAUSE>;
72                          | SEND ERROR <SUBSCRIPT>;
73                          | <ON CLAUSE>;
74                          | <ON CLAUSE> AND <SIGNAL CLAUSE>;
75                          | OFF ERROR <SUBSCRIPT>;
76                          | <% MACRO NAME>;
77                          | <% MACRO HEAD> <% MACRO ARG>);

78      <% MACRO HEAD> :: = <% MACRO NAME> (
79                           | <% MACRO HEAD> <% MACRO ARG> ,

80      <% MACRO ARG> :: = <NAME VAR>
81                         | <CONSTANT>

82      <BIT PRIM> :: = <BIT VAR>
83                    | <LABEL VAR>
84                    | <EVENT VAR>
85                    | <BIT CONST>
86                    | (<BIT EXP>)
87                    | <MODIFIED BIT FUNC>
88                    | <BIT INLINE DEF> <BLOCK BODY> <CLOSING>;
89                    | <SUBBIT HEAD> <EXPRESSION>)
90                    | <BIT FUNC HEAD> (<CALL LIST>)

91      <BIT FUNC HEAD> :: = <BIT FUNC>
92                           | BIT <SUB OR QUALIFIER>

93      <BIT CAT> :: = <BIT PRIM>
94                   | <BIT CAT> <CAT> <BIT PRIM>
95                   | <NOT> <BIT PRIM>
96                   | <BIT CAT> <CAT> <NOT> <BIT PRIM>

97      <BIT FACTOR> :: = <BIT CAT>
98                        | <BIT FACTOR> <AND> <BIT CAT>

99      <BIT EXP>  :: = <BIT FACTOR>
100                   | <BIT EXP> <OR> <BIT FACTOR>

101      <RELATIONAL OP> :: = =
102                           | <NOT> =
103                           | <
104                           | >
105                           | < =
106                           | > =
107                           | <NOT> <
108                           | <NOT> >

109      <COMPARISON> :: = <ARITH EXP> <RELATIONAL OP> <ARITH EXP>
110                       | <CHAR EXP> <RELATIONAL OP> <CHAR EXP>
111                       | <BIT CAT> <RELATIONAL OP> <BIT CAT>
```

```
112                    | <STRUCTURE EXP> <RELATIONAL OP> <STRUCTURE EXP>
113                    | <NAME EXP> <RELATIONAL OP> <NAME EXP>
114     <RELATIONAL FACTOR> :: = <REL PRIM>
115                            | <RELATIONAL FACTOR> <AND> <REL PRIM>
116     <RELATIONAL EXP> :: = <RELATIONAL FACTOR>
117                         | <RELATIONAL EXP> <OR> <RELATIONAL FACTOR>
118     <REL PRIM> :: = (1<RELATIONAL EXP>)
119                   | <NOT> (1<RELATIONAL EXP>)
120                   | <COMPARISON>
121     <CHAR PRIM> :: = <CHAR VAR>
122                    | <CHAR CONST>
123                    | <MODIFIED CHAR FUNC>
124                    | <CHAR INLINE DEF> <BLOCK BODY> <CLOSING>;
125                    | <CHAR FUNC HEAD> (<CALL LIST>)
126                    | (<CHAR EXP>)
127     <CHAR FUNC HEAD> :: = <CHAR FUNC>
128                         | CHARACTER <SUB OR QUALIFIER>
129     <SUB OR QUALIFIER> :: = <SUBSCRIPT>
130                           | <BIT QUALIFIER>
131     <CHAR EXP> :: = <CHAR PRIM>
132                   | <CHAR EXP> <CAT> <CHAR PRIM>
133                   | <CHAR EXP> <CAT> <ARITH EXP>
134                   | <ARITH EXP> <CAT> <ARITH EXP>
135                   | <ARITH EXP> <CAT> <CHAR PRIM>
136     <ASSIGNMENT> :: = <VARIABLE> <=1> <EXPRESSION>
137                     | <VARIABLE> , <ASSIGNMENT>
138     <IF STATEMENT> :: = <IF CLAUSE> <STATEMENT>
139                       | <TRUE PART> <STATEMENT>
140     <TRUE PART> :: = <IF CLAUSE> <BASIC STATEMENT> ELSE
141     <IF CLAUSE> :: = <IF> <RELATIONAL EXP> THEN
142                    | <IF> <BIT EXP> THEN
143     <IF> :: =  IF
144     <DO GROUP HEAD> :: = DO;
145                        | DO <FOR LIST>;
146                        | DO <FOR LIST> <WHILE CLAUSE>;
147                        | DO <WHILE CLAUSE>;
148                        | DO CASE <ARITH EXP>;
149                        | <CASE ELSE> <STATEMENT>
150                        | <DO GROUP HEAD> <ANY STATEMENT>
151                        | <DO GROUP HEAD> <TEMPORARY STMT>
152     <CASE ELSE> :: = DO CASE <ARITH EXP>; ELSE
153     <WHILE KEY> :: = WHILE
154                    | UNTIL
155     <WHILE CLAUSE> :: = <WHILE KEY> <BIT EXP>
156                       | <WHILE KEY> <RELATIONAL EXP>
157     <FOR LIST> :: = <FOR KEY> <ARITH EXP> <ITERATION CONTROL>
158                   | <FOR KEY> <ITERATION BODY>
159     <ITERATION BODY> :: = <ARITH EXP>
160                         | <ITERATION BODY> , <ARITH EXP>
161     <ITERATION CONTROL> :: = TO <ARITH EXP>
162                            | TO <ARITH EXP> BY <ARITH EXP>
```

```
163     <FOR KEY> :: = FOR <ARITH VAR> =
164                  | FOR TEMPORARY <IDENTIFIER> =
165     <ENDING> :: = END
166                 | END <LABEL>
167                 | <LABEL DEFINITION> <ENDING>
168     <ON PHRASE> :: = ON ERROR <SUBSCRIPT>
169     <ON CLAUSE> :: = ON ERROR <SUBSCRIPT> SYSTEM
170                    | ON ERROR <SUBSCRIPT> IGNORE
171     <SIGNAL CLAUSE> :: = SET <EVENT VAR>
172                        | RESET <EVENT VAR>
173                        | SIGNAL <EVENT VAR>
174     <FILE EXP> :: = <FILE HEAD> , <ARITH EXP>)
175     <FILE HEAD> :: = FILE (<NUMBER>
176     <CALL KEY> :: = CALL <LABEL VAR>
177     <CALL LIST> :: = <LIST EXP>
178                    | <CALL LIST> , <LIST EXP>
179     <CALL ASSIGN LIST> :: = <VARIABLE>
180                           | <CALL ASSIGN LIST> , <VARIABLE>
181     <EXPRESSION> :: = <ARITH EXP>
182                     | <BIT EXP>
183                     | <CHAR EXP>
184                     | <STRUCTURE EXP>
185                     | <NAME EXP>
186     <STRUCTURE EXP> :: = <STRUCTURE VAR>
187                        | <MODIFIED STRUCT FUNC>
188                        | <STRUC INLINE DEF> <BLOCK BODY> <CLOSING>;
189                        | <STRUCT FUNC HEAD> (<CALL LIST>)
190     <STRUCT FUNC HEAD> :: = <STRUCT FUNC>
191     <LIST EXP> :: = <EXPRESSION>
192                   | <ARITH EXP> # <EXPRESSION>
193     <VARIABLE> :: = <ARITH VAR>
194                   | <STRUCTURE VAR>
195                   | <BIT VAR>
196                   | <EVENT VAR>
197                   | <SUBBIT HEAD> <VARIABLE>)
198                   | <CHAR VAR>
199                   | <NAME KEY> (<NAME VAR>)
200     <NAME VAR> :: = <VARIABLE>
201                   | <LABEL VAR>
202                   | <MODIFIED ARITH FUNC>
203                   | <MODIFIED BIT FUNC>
204                   | <MODIFIED CHAR FUNC>
205                   | <MODIFIED STRUCT FUNC>
206     <NAME EXP> :: = <NAME KEY> (<NAME VAR>)
207                   | NULL
208                   | <NAME KEY> (NULL)
209     <NAME KEY> :: = NAME
210     <LABEL VAR> :: = <PREFIX> <LABEL> <SUBSCRIPT>
211     <MODIFIED ARITH FUNC> :: = <PREFIX> <NO ARG ARITH FUNC> <SUBSCRIPT>
212     <MODIFIED BIT FUNC> :: = <PREFIX> <NO ARG BIT FUNC> <SUBSCRIPT>
213     <MODIFIED CHAR FUNC> :: = <PREFIX> <NO ARG CHAR FUNC> <SUBSCRIPT >
214     <MODIFIED STRUCT FUNC> :: = <PREFIX> <NO ARG STRUCT FUNC> <SUBSCRIPT>
```

```
215     <STRUCTURE VAR> :: = <QUAL STRUCT> <SUBSCRIPT>
216     <ARITH VAR> :: = <PREFIX> <ARITH ID> <SUBSCRIPT>
217     <CHAR VAR> :: = <PREFIX> <CHAR ID> <SUBSCRIPT>
218     <BIT VAR> :: = <PREFIX> <BIT ID> <SUBSCRIPT>
219     <EVENT VAR> :: = <PREFIX> <EVENT> <SUBSCRIPT>
220     <QUAL STRUCT> :: = <STRUCTURE ID>
221                      | <QUAL STRUCT> . <STRUCTURE ID>
222     <PREFIX> :: = <EMPTY>
223                 | <QUAL STRUCT> .
224     <SUBBIT HEAD> :: = <SUBBIT KEY> <SUBSCRIPT> (
225     <SUBBIT KEY> :: = SUBBIT
226     <SUBSCRIPT> :: = <SUB HEAD>)
227                    | <QUALIFIER>
228                    | <$> <NUMBER>
229                    | <$> <ARITH VAR>
230                    | <EMPTY>
231     <SUB START> :: = <$>(
232                    | <$> (@<PREC SPEC> ,
233                    | <SUB HEAD>;
234                    | <SUB HEAD> :
235                    | <SUB HEAD>,
236     <SUB HEAD> :: = <SUB START>
237                   | <SUB START> <SUB>
238     <SUB> :: = <SUB EXP>
239              | *
240              | <SUB RUN HEAD> <SUB EXP>
241              | <ARITH EXP> AT <SUB EXP>
242     <SUB RUN HEAD> :: = <SUB EXP> TO
243     <SUB EXP> :: = <ARITH EXP>
244                  | <# EXPRESSION>
245     <# EXPRESSION> :: = #
246                       | <# EXPRESSION> + <TERM>
247                       | <# EXPRESSION> -1 <TERM>
248     <=1> :: = =
249     <$> :: = $
250     <AND> :: = &
251              | AND
252     <OR>  : =|
253              | OR
254     <NOT> :: =  ¬
255              | NOT
256     <CAT> :: = ||
257              | CAT
258     <QUALIFIER> :: = <$> (@<PREC SPEC>)
259                    | <$> (<SCALE HEAD><ARITH EXP>)
260                    | <$> (@<PREC SPEC>, <SCALE HEAD><ARITH EXP>)
261    <SCALE HEAD> :: = @
262                      | @ @
263     <BIT QUALIFIER> :: = <$(@<RADIX>)
```

```
264     <RADIX> :: = HEX
265                | OCT
266                | BIN
267                | DEC
268     <BIT CONST HEAD> :: = <RADIX>
269                         | <RADIX> (<NUMBER>)
270     <BIT CONST> :: = <BIT CONST HEAD> <CHAR STRING>
271                    | TRUE
272                    | FALSE
273                    | ON
274                    | OFF
275     <CHAR CONST> :: = <CHAR STRING>
276                     | CHAR (<NUMBER>) <CHAR STRING>
277     <IO CONTROL> :: = SKIP (<ARITH EXP>)
278                     | TAB (<ARITH EXP>)
279                     | COLUMN (<ARITH EXP>)
280                     | LINE (<ARITH EXP>)
281                     | PAGE (<ARITH EXP>)
282     <READ PHRASE> :: = <READ KEY> <READ ARG>
283                      | <READ PHRASE> , <READ ARG>
284     <WRITE PHRASE> :: = <WRITE KEY> <WRITE ARG>
285                       | <WRITE PHRASE> , <WRITE ARG>
286     <READ ARG> :: = <VARIABLE>
287                   | <IO CONTROL>
288     <WRITE ARG> :: = <EXPRESSION>
289                    | <IO CONTROL>
290     <READ KEY> :: = READ (<NUMBER>)
291                   | READALL (<NUMBER>)
292     <WRITE KEY> :: = WRITE (<NUMBER>)
293     <BLOCK DEFINITION> :: = <BLOCK STMT> <BLOCK BODY> <CLOSING>;
294     <BLOCK BODY> :: = <EMPTY>
295                     | <DECLARE GROUP>
296                     | <BLOCK BODY> <ANY STATEMENT>
297     <ARITH INLINE DEF> :: = FUNCTION <ARITH SPEC>;
298                           | FUNCTION;
299     <BIT INLINE DEF> :: = FUNCTION <BIT SPEC>;
300     <CHAR INLINE DEF> :: = FUNCTION <CHAR SPEC>;
301     <STRUC INLINE DEF> :: = FUNCTION <STRUCT SPEC>;
302     <BLOCK STMT> :: = <BLOCK STMT TOP>;
303     <BLOCK STMT TOP> :: = <BLOCK STMT TOP> ACCESS
304                         | <BLOCK STMT TOP> RIGID
305                         | <BLOCK STMT HEAD>
306                         | <BLOCK STMT HEAD> EXCLUSIVE
307                         | <BLOCK STMT HEAD> REENTRANT
308     <LABEL DEFINITION> :: = <LABEL>:
309     <LABEL EXTERNAL> :: = <LABEL DEFINITION>
310                         | <LABEL DEFINITION> EXTERNAL
311     <BLOCK STMT HEAD> :: = <LABEL EXTERNAL> PROGRAM
312                          | <LABEL EXTERNAL> COMPOOL
313                          | <LABEL DEFINITION> TASK
314                          | <LABEL DEFINITION> UPDATE
315                          | UPDATE
316                          | <FUNCTION NAME>
317                          | <FUNCTION NAME> <FUNC STMT BODY>
```

```
318                          | <PROCEDURE NAME>
319                          | <PROCEDURE NAME> <PROC STMT BODY>
320     <FUNCTION NAME> :: = <LABEL EXTERNAL> FUNCTION
321     <PROCEDURE NAME> :: = <LABEL EXTERNAL> PROCEDURE
322     <FUNC STMT BODY> :: = <PARAMETER LIST>
323                          | <TYPE SPEC>
324                          | <PARAMETER LIST> <TYPE SPEC>
325     <PROC STMT BODY> :: = <PARAMETER LIST>
326                          | <ASSIGN LIST>
327                          | <PARAMETER LIST> <ASSIGN LIST>
328     <PARAMETER LIST> :: = <PARAMETER HEAD> <IDENTIFIER>)
329     <PARAMETER HEAD> :: = (
330                          | <PARAMETER HEAD> <IDENTIFIER>,
331     <ASSIGN LIST> :: = <ASSIGN> <PARAMETER LIST>
332     <ASSIGN> :: = ASSIGN
333     <DECLARE ELEMENT> :: = <DECLARE STATEMENT>
334                           | <REPLACE STMT>;
335                           | <STRUCTURE STMT>
336                           | EQUATE EXTERNAL <IDENTIFIER> TO <VARIABLE>;
337     <REPLACE STMT> :: = REPLACE <REPLACE HEAD> BY <TEXT>
338     <REPLACE HEAD> :: = <IDENTIFIER>
339                       | <IDENTIFIER> (<ARG LIST>)
340     <ARG LIST> :: = <IDENTIFIER>
341                   | <ARG LIST> , <IDENTIFIER>
342     <TEMPORARY STMT> :: = TEMPORARY <DECLARE BODY>;
343     <DECLARE STATEMENT> :: = DECLARE <DECLARE BODY>;
344     <DECLARE BODY> :: = <DECLARATION LIST>
345                       | <ATTRIBUTES> , <DECLARATION LIST>
346     <DECLARATION LIST> :: = <DECLARATION>
347                           | <DCL LIST,> <DECLARATION>
348     <DCL LIST ,> :: = <DECLARATION LIST>,
349     <DECLARE GROUP> :: = <DECLARE ELEMENT>
350                        | <DECLARE GROUP> <DECLARE ELEMENT>
351     <STRUCTURE STMT> :: = STRUCTURE <STRUCT STMT HEAD> <STRUCT STMT TAIL>
352     <STRUCT STMT HEAD> :: = <IDENTIFIER> : <LEVEL>
353                           | <IDENTIFIER> <MINOR ATTR LIST>: <LEVEL>
354                           | <STRUCT STMT HEAD> <DECLARATION> , <LEVEL>
355     <STRUCT STMT TAIL> :: = <DECLARATION>;
356     <STRUCT SPEC> :: = <STRUCT TEMPLATE> <STRUCT SPEC BODY>
357     <STRUCT SPEC BODY> :: = - STRUCTURE
358                           | <STRUCT SPEC HEAD> <LITERAL EXP OR *>)
359     <STRUCT SPEC HEAD> :: = - STRUCTURE (
360     <DECLARATION> :: = <NAME ID>
361                      | <NAME ID> <ATTRIBUTES>
362     <NAME ID> :: = <IDENTIFIER>
363                  | <IDENTIFIER> NAME
364     <ATTRIBUTES> :: = <ARRAY SPEC> <TYPE & MINOR ATTR>
365                     | <ARRAY SPEC>
366                     | <TYPE & MINOR ATTR>
```

```
<ARRAY SPEC> :: = <ARRAY HEAD> <LITERAL EXP OR *>)
                | FUNCTION
                | PROCEDURE
                | PROGRAM
                | TASK
<ARRAY HEAD> :: = ARRAY (
                | <ARRAY HEAD> <LITERAL EXP OR *>,
<TYPE & MINOR ATTR> :: = <TYPE SPEC>
                       | <TYPE SPEC> <MINOR ATTR LIST>
                       | <MINOR ATTR LIST>
<TYPE SPEC> :: = <STRUCT SPEC>
               | <BIT SPEC>
               | <CHAR SPEC>
               | <ARITH SPEC>
               | EVENT
<BIT SPEC> :: = BOOLEAN
              | BIT (<LITERAL EXP OR *>)
<CHAR SPEC> :: = CHARACTER (<LITERAL EXP OR *>)
<ARITH SPEC> :: = <PREC SPEC>
                | <SQ DQ NAME>
                | <SQ DQ NAME> <PREC SPEC>
<SQ DQ NAME> :: = <DOUBLY QUAL NAME HEAD> <LITERAL EXP OR *>)
                | INTEGER
                | SCALAR
                | VECTOR
                | MATRIX
<DOUBLY QUAL NAME HEAD> :: = VECTOR (
                                | MATRIX (<LITERAL EXP OR *> ,
<LITERAL EXP OR *>  :: = <ARITH EXP>
                       | *
<PREC SPEC> :: = SINGLE
               | DOUBLE
<MINOR ATTR LIST>  :: = <MINOR ATTRIBUTE>
                      | <MINOR ATTR LIST> <MINOR ATTRIBUTE>
<MINOR ATTRIBUTE> :: =  STATIC
                      | AUTOMATIC
                      | DENSE
                      | ALIGNED
                      | ACCESS
                      | LOCK (<LITERAL EXP OR *>)
                      | REMOTE
                      | RIGID
                      | <INIT/CONST HEAD> <REPEATED CONSTANT>)
                      | <INIT/CONST HEAD> *)
                      | LATCHED
                      | NONHAL (<LEVEL>)
<INIT/CONST HEAD> :: = INITIAL (
                      | CONSTANT (
                      | <INIT/CONST HEAD> <REPEATED CONSTATN> ,
<REPEATED CONSTANT> :: = EXPRESSION
                       | <REPEAT HEAD> <VARIABLE>
                       | <REPEAT HEAD> <CONSTANT>
                       | <NESTED REPEAT HEAD> <REPEATED CONSTANT>)
                       | <REPEAT HEAD>
<REPEAT HEAD> :: = <ARITH EXP> #
<NESTED REPEAT HEAD> :: = <REPEAT HEAD> (
                        | <NESTED REPEAT HEAD> <REPEATED CONSTANT> ,
```

```
424     <CONSTANT> :: = <NUMBER>
425                   | <COMPOUND NUMBER>
426                   | <BIT CONST>
427                   | <CHAR CONST>

428     <NUMBER> :: = <SIMPLE NUMBER>
429                 | <LEVEL>

430     <CLOSING> :: = CLOSE
431                  | CLOSE <LABEL>
432                  | <LABEL DEFINITION> <CLOSING>

433     <TERMINATOR> :: = TERMINATE
434                     | CANCEL

435     <TERMINATE LIST> :: = <LABEL VAR>
436                         | <TERMINATE LIST> , <LABEL VAR>

437     <WAIT KEY> :: = WAIT

438     <SCHEDULE HEAD> :: = SCHEDULE <LABEL VAR>
439                        | <SCHEDULE HEAD> AT <ARITH EXP>
440                        | <SCHEDULE HEAD> IN <ARITH EXP>
441                        | <SCHEDULE HEAD> ON <BIT EXP>

442     <SCHEDULE PHRASE> :: = <SCHEDULE HEAD>
443                          | <SCHEDULE HEAD> PRIORITY(<ARITH EXP>)
444                          | <SCHEDULE PHRASE> DEPENDENT

445     <SCHEDULE CONTROL> :: = <STOPPING>
446                           | <TIMING>
447                           | <TIMING> <STOPPING>

448     <TIMING> :: = <REPEAT> EVERY <ARITH EXP>
449                 | <REPEAT> AFTER <ARITH EXP>
450                 | <REPEAT>

451     <REPEAT> :: = , REPEAT

452     <STOPPING> :: = <WHILE KEY> <ARITH EXP>
453                   | <WHILE KEY> <BIT EXP>
```

This page is intentionally left blank.

Appendix H - SUMMARY OF OPERATORS

This section contains a series of tables which explicitly summarize the possible arithmetic, bit, character, and conditional operators used in forming expressions in the HAL/S language.

The information found in this appendix has been abstracted from Chapter 6 of this Specification.

H.1 ARITHMETIC OPERATORS[18]

OPERATORS	NAME	ARITHMETIC PRECEDENCE	FORM	COMMENTS
**	Exponentiation	1	x**x m**i m**0 m**-i m**T	Ordinary exponentiation Repeated Multiplication Identity matrix Repeated mult. of inverse Transpose of matrix
(blank) < >	Product	2	m m m v v m v v x m m x v x x v x x	matrix-matrix product matrix-vector product vector-matrix product outer product scalar or integer product with matrix/vector scalar or integer product with scalar or integer
*	Cross Product	3	v*v	cross product of two 3-vectors
.	Dot Product	4	v.v	dot product of two vectors
/	Division	5	m/x v/x x/x	division of left-hand term by scalar or integer
+ -	Addition Subtraction	6	x+x m+m v+v x-x m-m v-v +x +m +v -x -m -v	Algebraic addition or subtraction; binary plus and minus
	The following abbreviations apply: i = positive integer literal x = scalar or integer M = matrix v = vector			

18. Note that this table contains information found in Section 6.1.1.

H.2 CHARACTER OPERATOR[19]

OPERATOR	NAME	FORM
||	concatenation	re||sult → result

H.3 BIT OPERATORS[20]

OPERATOR	NAME	BIT OPERATOR PRECEDENCE	FORM	COMMENTS
|| CAT	concatenation	1	B||B	11101 || 010 →11101010
& AND	logical product	2	B&B	Parallel operation bit by bit
| OR	logical sum	3	B|B	Parallel operation bit by bit
¬ NOT	logical complement	Highest implied by syntax	B	Parallel operation bit by bit
The following abbreviations apply: B= bit string or BOOLEAN				

H.4 CONDITIONAL AND EVENT OPERATORS[21]

OPERATOR	NAME	CONDITIONAL PRECEDENCE	FORM	COMMENTS
& AND	logical product	1	C&C C AND C	True if both “C”s true
| OR	logical sum	2	C|C C OR C	True if either “C” is true
¬ NOT	logical complement	Highest implied by syntax	¬C NOT C	Operand
The following abbreviations apply: “C”= any conditional operand.				

19. Note that this table contains information found in Section 6.1.3.
20. Note that this table contains information found in Section 6.1.2.
21. Note that this table contains information found in Sections 6.2 and 6.3.

H.5 COMPARISON OPERATORS[22]

OPERATOR	USE	COMMENTS
>	A > B	- Magnitude comparison: apply only to unarrayed scalar and integer data A and B.
>=	A >=B	
¬<	A ¬< B	
NOT<		
<	A < B	
<=	A <= B	
¬>	A ¬>B	
NOT>		
=	A = B	Equality/inequality for general data A and B.
NOT=	A ¬ = B	
¬=		

22. Note that this table contains information found in Section 6.2.

This page is intentionally left blank.

Appendix I - %MACROS

The specific details of %macro operation as well as the %macros available are implementation dependent. A generic description of %macro syntax can be found in Section 11.2 of this document.

Individual implementations of the HAL/S language may contain %macro capabilities. The documentation for each implementation (such as a User's Manual) will contain the detailed descriptions of the available %macros.

This page is intentionally left blank.

Appendix J - CHANGE HISTORY

Change History				
Revision	Release	Date	Change Authority	Sections Changed
01		01/12/81		Title Page, 2-7 - 2-9, 4-2, 4-15, 4-18 - 4-19, 4-22 - 4-23, 4-27 - 4-29, 5-9, 5-12, 5-15 - 5.17.2, 5-19, 6-3 - 6-7, 6-17, 6-25, 6-28 - 6-30, 6-32, 6-34 - 6-36, 6-38 - 6-40, 7-6 - 7-7, 7-11 - 7-12, 7-14, 7-20 - 7-23, 8-6 - 8-7, 8-11, 11-4, 11-6, 11-8, 11-14, 11-17, 11-19, B-1, C-1 - C-8, C-10, D-1 - D-3, E-2 - E-3, F-1, G-1 - G-11, H-2
02	25.0/9.0	09/03/93		Title Page, Section 6.2.4 (p.6-20)
03	27.0/11.0	05/22/96		Total Reprint
04	27.1/11.1	07/01/96		pp. - 6-3, 6-4, 6-13, 6-21, 11-2, D-1, G-1, G-6 - G-7, G-10
05	28.0/12.0	08/22/97		Total reprint to bring to HAL/S documentation standards and HTML compatibility.
			CR12709	2.3.4 - p. 2-6 4.2.1 - pp. 4-3, 4-4 4.2.3 - p. 4-5 4.2.4 - p. 4-5 4.5 - P. 4-13 6.1.1 - p. 6-4 6.1.2 - pp. 6-6, 6-7, 6-11 6.2.1 - pp. 6-12, p. 6-13 6.2.2 - p. 6-13 6.2.4 - pp. 6-15, 6-17 6.3 - P. 6-7 11.2.1 - p. 11-2 App. C - p. C-4
			CR12712	2.3.3 - p. 2-4 2.3.4 - p. 2-5 4.0 - p. 4-1 4.5 - pp. 4-11, 4-13 4.7 - pp. 4-15, 4-16 4.8 - pp. 4-19, 4-20 5.3.1 - p. 5-5 5.3.5 - p. 5-11 5.4.1 - p. 5-13 5.5 - P. 5-14 6.1.1 - pp. 6-2, 6-3, 6-4, 6-5 6.2.1 - pp. 6-13 6.4 - p. 6-19 6.5.1 - pp. 6-20, 6-21, 6-22, 6-23 6.5.2 - p. 6-24

				6.5.3	- p. 6-25
				6.5.4	- p. 6-26
				6.5.5	- p. 6-28
				6.6	- p. 6-28
				6.7	- Deleted
				7.3	- p. 7.3
				7.4	- p. 7-6
				7.5	- p. 7-8
				7.6.4	- p. 7-11
				7.6.5	- p. 7-12
				8.3	- p. 8-4
				8.6	- p. 8-7
				11.2.1	- p. 11-3
				11.2.2	- p. 11-4
				11.2.3	- p. 11-5
				11.3.1	- p. 11-10
				11.4.1	- pp. 11-12, 11-13
				App. A.2	- p. A-5
				App. A.3	- pp. A-6, A-7, A-8
				App. B	- p. B-1
				App. C	- pp. C-1, C-2, C-3, C-4, C-5
				App. D	- pp. D-1, D-2
				App. E	- pp. E-1, E-2
				App. F	- p. F-1
				App. G	- pp. G-1, G-5, G-6, G-9, G-10
				App. H	- p. H-1
				Index	- pp. Index-1- Index-10

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

This is the Last Page of this Document

DRD NUMBER: 1.4.3.8-a DOCUMENT NUMBER: USA003088

TITLE: HAL/S Language Specification

NASA-JSC

*BV	N. Moses
MS4	D. Stamper
EV111	EV Library (D. Wall)

USA-Houston

*USH-121G	SFOC Technical Library
USH-634G	Abel Puente
USH-64A6X	L.W. Wingo
USH-633L	Anita Senviel
USH-633L	Benjamin L. Peterson
USH-633L	Cory L. Driskill
USH-633L	Judy M. Hardin
USH-633L	Mark E. Lading
USH-633L	Quinn L. Larson
USH-633L	James T. Tidwell
USH-633L	Vicente Aguilar
USH-633L	Betty A. Pages
USH-633L	Jeremy C. Battan
USH-633L	George H. Ashworth
USH-634L	Mark Caronna
USH-634L	Burk J. Royer
*USH-635L	Joy C. King
USH-635L	Ling J. Kuo
USH-635L	Trang K. Nguyen
USH-635L	Billy L. Pate
USH-635L	Karen H. Pham
*USH-635L	Dan A. Strauss
USH-635L	Pete Koester
USH-632L	Renne Siewers
*USH-635L	Barbara Whitfield (2)

Boeing

HS1-40 B. Frere
blake.a.frere@boeing.com

* Denotes hard copy

Submit NASA distribution changes, including initiator's name and phone number, to JSC Data Management/BV or call 281-244-8506. Submit USA distribution changes to USA Data Management/USH-121E or via e-mail to *usadm@usa-spaceops.com*. Most documents are available electronically via USA Intranet Web (*usa1.unitedspacealliance.com*), Space Flight Operations Contract (SFOC), SFOC Data and Deliverables.

Indicates hardcopy