Report No. 75-0043
Contract No. NAS8-26990

(NASA-CR-144100) HAL/SM LANGUAGE SPECIFICATION (M&S Computing, Inc.) 253 p HC [redacted] CSCL 09B

N76-14845

Unclas
G3/61 06781

HAL/SM LANGUAGE SPECIFICATION

November 21, 1975

Prepared for:
George C. Marshall Space Flight Center
NASA
Marshall Space Flight Center, AL 35812

M&S COMPUTING, INC.
Post Office Box 5183
Huntsville, AL 35805

PREFACE

This document constitutes the formal HAL/SM Language Specification, its scope being limited to the essentials of HAL/SM 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/SM 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/SM programming system.

Prepared by:

G. P. Williams, Jr.

C. Ross

Approved by:

[illegible]

J. L. Pruitt

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

(BLANK)

1. INTRODUCTION

HAL/S is a programming language developed by Intermetrics, Inc., for the flight software of the NASA Space Shuttle program. HAL/S is intended to satisfy virtually all of the flight software requirements of the Space Shuttle. 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/SM is a dialect of the original HAL language previously developed by Intermetrics (see Reference 1) and of the HAL/S dialect, also developed by Intermetrics (see Reference 2). Changes have been incorporated to adapt the language to the MOSS environment.

HAL/SM 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.

Data Types and Computations

HAL/SM 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/SM 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/SM is a real time control language. Defined blocks of code called 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/SM 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/SM contains a number of features especially designed to facilitate its application 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/SM 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 task blocks can be executed together and communicate through one or more centrally managed and highly visible data pools. In a real time environment, HAL/SM couples these precautions with locking mechanisms preventing the uncontrolled usage of sensitive data or areas of code.

The formal Specification of HAL/SM 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/SM is presented in Section 3. Data declaration and referencing are presented in Sections 4 and 5 respectively. Section 6 is devoted 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, and so on. Section 8 deals with real time programming. Section 9 explains the HAL/SM error recovery system and Section 10 the HAL/SM I/O capability.

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

2. SYNTAX DIAGRAMS AND HAL/SM PRIMITIVES

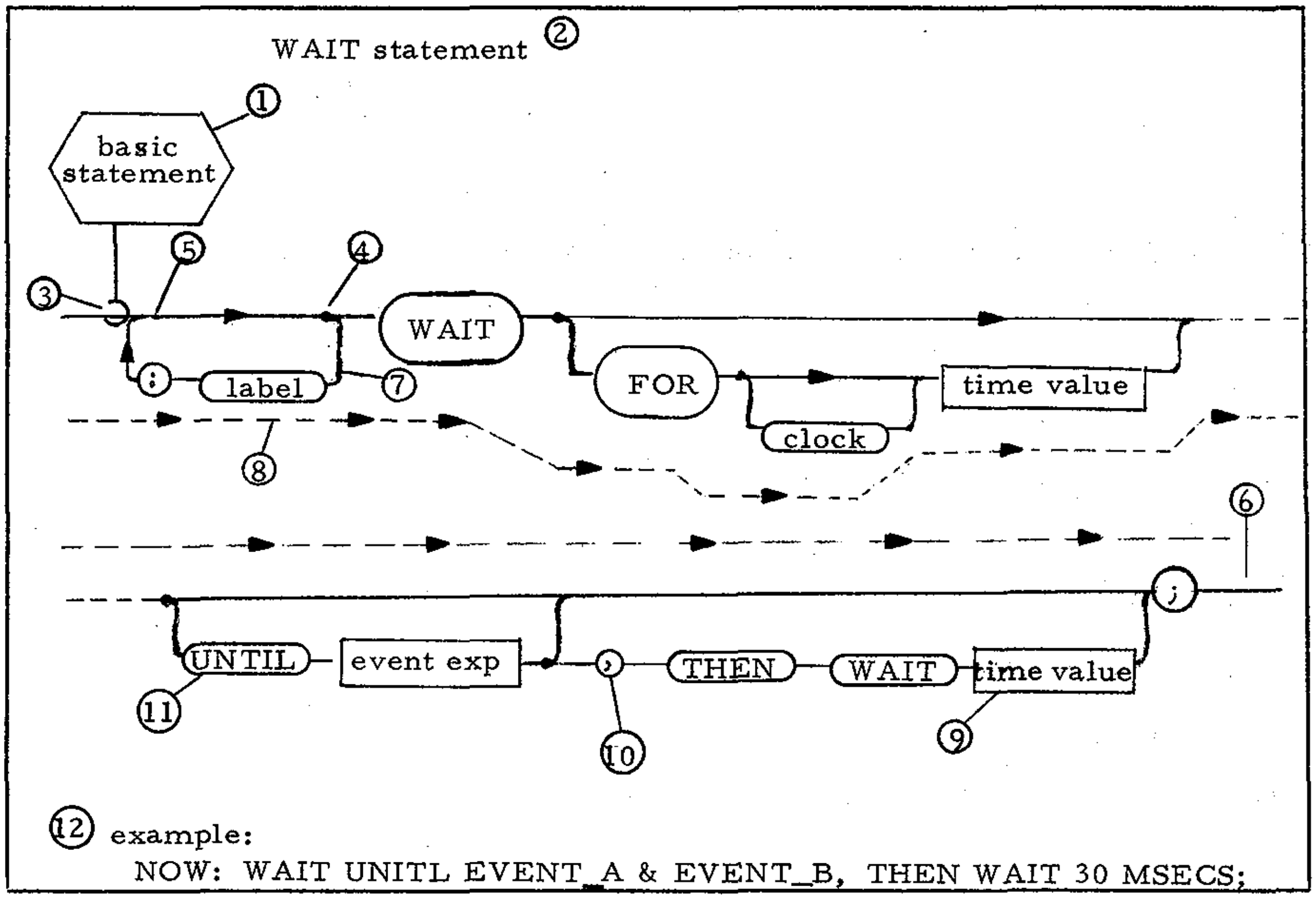
In this Specification, the syntax of the HAL/SM language is represented in the form of syntax diagrams. These are to be read in conjunction with the associated sets of semantic rules. Sometimes the semantic rules modify or restrict the meaning inherent in the syntax diagrams. Together the two provide 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/SM "primitives."

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

2.1 The HAL/SM 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/SM language. The syntax diagrams are, however, not meant to be viewed as constituting a "working" grammar (that is, as 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.



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 ② 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 samples 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 the dotted line ⑧ generates the sequence "WAIT FOR <clock> <time value>;" (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/SM primitives. Circular boxes, such as ⑩ , contain special characters (see Section 2.2).

7. In the text accompanying the syntax diagrams, boxes containing lower case names are represented by enclosing the names in the delimiters < >. Thus box ⑨ becomes <time value> . Upper case names are reserved words of the language.

8. The example given at ⑫ is an example of HAL/SM 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/SM Character Set

The HAL/SM character set consists of the 52 upper and lower case alphabetic 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 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.

alphabetic		alphabetic		special characters	
A	identifiers, literals, reserved words	j	literals	|	operators
B		k		¬	
C		l		&	
D		m		#	
E		n		@	
F		o		$	
G		p		<	operators, ERP designators
H		q		>	
I		r		=	
J		s		*	
K		t		/	
L		u		+	
M		v		−	
N		w		.	
O		x		;	separators
P		y		:	
Q		z		,	
R		**pseudo-alphabetic**		(blank)	separators ERP designators
S		-	identifiers	(	delimiters
T		¢	text generation excape	)	
U		**numeric**		'	
V		0	identifiers literals	"	
W		1		**additional extended-set symbols**	
X		2		[	
Y		3		]	
Z		4		{	
		5		}	
a		6		!	
b		7		?	
c		8			
d		9			
e					
f					
g					
h					
i					

2.3 HAL/SM Primitives

HAL/SM syntax diagrams ultimately express all syntactical elements in terms of a small number of special characters and predefined primitives. Primitives are constructed from the characters comprising the HAL/SM 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. They are constructed from alphabetic characters alone. Reserved words fall into two categories: keywords and built-in function names. In the syntax diagrams, and in the accompanying text, reserved words are indicated by upper case characters.

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:

- o the total number of characters must not exceed 32,
- o the first character must be alphabetic,
- o any character except the first may be alphabetic or numeric, and
- o 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 syntactical names, as appropriate:

<label>

< § var name>

where: §	≡	arith (arithmetic)
		char (character)
		bit
		event
		structure

<flag>

<template name>

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 ERP Designators

An ERP designator is a name used to specify an External Reference Point. The ERP designator must be a duplicate of an ERP name defined in the Measurement and Control Definition (M&CD). The ERP designator is syntactically known as an <erp designator>. The following rules must be satisfied by the <erp designator>.

- o The total number of characters must not exceed 31;
- o The first and last characters must be angle brackets; the first character being "<" (less than) and the last character being ">" (greater than);
- o Any character except the first and last may be alphabetic or numberic; and
- o Any character except the first and last may be any of the following symbols:

*	(asterisk)	-	(minus)
ƀ	(blank)	.	(period)
,	(comma)	+	(plus)
=	(equals)	/	(slash)

2.3.4 Literals

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

<u>Rules for Arithmetic Literals</u>

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:

   $\pm$ ddddddd. dddddddd <exponents>

   where d = decimal digit.

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

3. The form of any of the < exponents > may be

B <power>	$\sim 2^{<\text{power}>}$
E <power>	$\sim 10^{<\text{power}>}$
H <power>	$\sim 16^{<\text{power}>}$

   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'

   where o = octal digit

   HEX < repetition> 'hhhhhhh'

   where h = hecadecimal digit

DEC <repetition> 'ddddddd'

where 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:

   OPEN ≡ TRUE ≡ ON ≡ BIN'1'

   CLOSED ≡ 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/SM extended character set. The < repetition> consists of the 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/SM extended character set. A "¢" is defined to be the "escape" character within this context. In accordance with an implementation dependent mapping scheme, HAL/SM characters will be assigned alternate character values. Inclusion of these alternate values in a string literal is achieved by preceding the appropriate HAL/SM 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/SM character to be the character "¢".

Examples:

' '

'ONE TWO THREE'

'DOG' 'S'

'AB¢AD'

'AB¢¢AD'

} The implication is that ¢A and ¢¢A have been defined as alternate characters.

Rules for Dimensioned Literals

1. Literals which are considered as dimensioned are denoted syntactically by <dim literal>.

2. The form of dimensioned literals is:

   <number> <dimension>

   where <dimension> is a set of engineering units found in Table 2-1.

   Examples:

   10V
   4MOHM

Rules for Time Literals

1. Literals which represent time values are denoted syntactically by <time literal>.

2. They have one of the forms shown below:

   <number> DAY or <number> DAYS

   <number> HR or <number> HRS

   <number> MIN or <number> MINS

   <number> SEC or <number> SECS

   <number> MSEC or <number> MSECS

Engineering Units

FUNCTION TYPE	BASIC UNIT	$X10^0$	$X10^3$	$X10^6$	$X10^{-3}$	$X10^{-6}$	$X10^{-12}$
volts dc	volts	VDC					
volts ac/dc	volt	V	KV		MV	UV	
current ac/dc	ampere	Z			MA	UA	
	hertz	HZ	KHZ	MHZ			
frequency	pulses per second	PPS	KPPS				
time	day	DAY					
	hour	HR					
	minute	MIN					
	second	SEC			MSEC	USEC	
resistance	ohm	OHM	KOHM	MOHM			
inductance	henry	H			MH	UH	
capacitance	farad	FD				UFD	PFD
power	watt	W	KW		MW	UW	
	voltage, current or power	VAR, DBW, DB	KVA KVAR		DBM		
ratio	percent	PCT					
pressure	pounds per square inch	PSIG PSIA PSI					
	millimeters of mercury	MMHG					
	inches of mercury	INHG					
	millibars	MB					
distance	inch	IN					
	foot	FT					
	meter	M	KM		MM		
	nautical mile	NM					
velocity	feet per second	FT/ SEC					
	meters per second	M/ SEC					
	knot	KT					
	mach no.	MACH					
angle	degree	DEG					
	arcmin	ARCMIN					
	arcsec	ARCSEC					
	radian	RAD			MRAD		
	revolution	REV					
temperature	degrees centigrade	DEGC					
	degrees fahrenheit	DEGF					
acceleration	meters/sec/ sec	M/S/S					
	feet/sec/sec	F/S/S					
mass	grams	G					
flowrate	gallons/minute	GPM					
	cubic feet/min	CFM					
	pounds/hour	LB/HR					

Table 2-1

<number> DAYS <number> HRS <number> MINS <number>

SECS <number> MSECS or any substitution of the above forms in this order.

When any <number> is zero, the associated unit may be omitted.

Examples:

3 DAYS
10 MINS 30 SECS
1 DAY 20 HRS 10 MINS 22 SECS

2.3.5 Display Control Word Value List

The display control word value list is a list of display options used to define a display control word (a word which identifies how C&D data is to be displayed). To form the display control word value list the display options are specified in the following order:

<color> ,

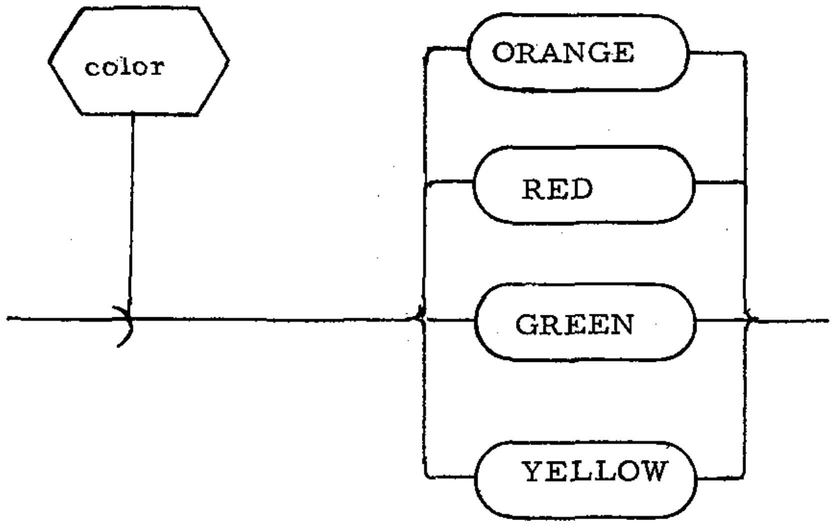
<character size> ,

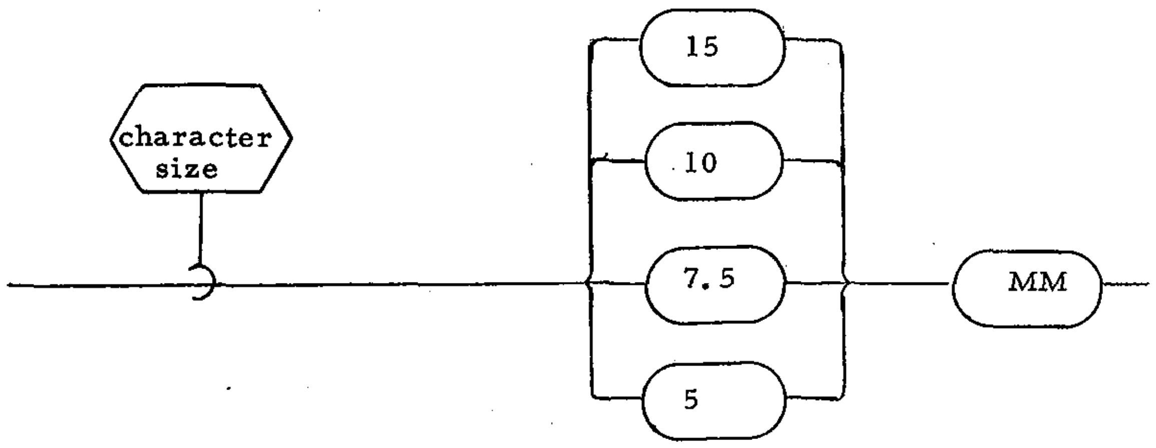
<blink data> ,

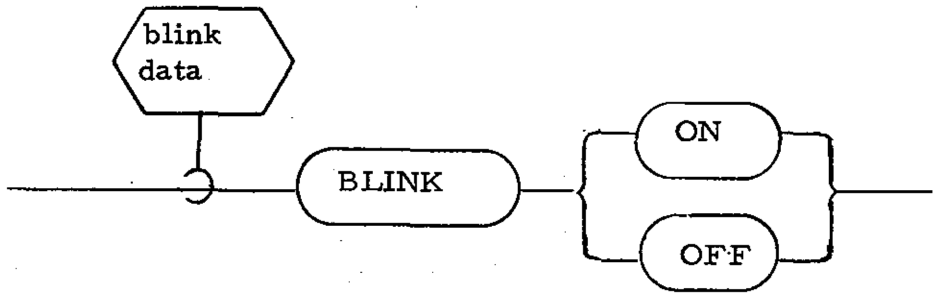
<intensity> , and

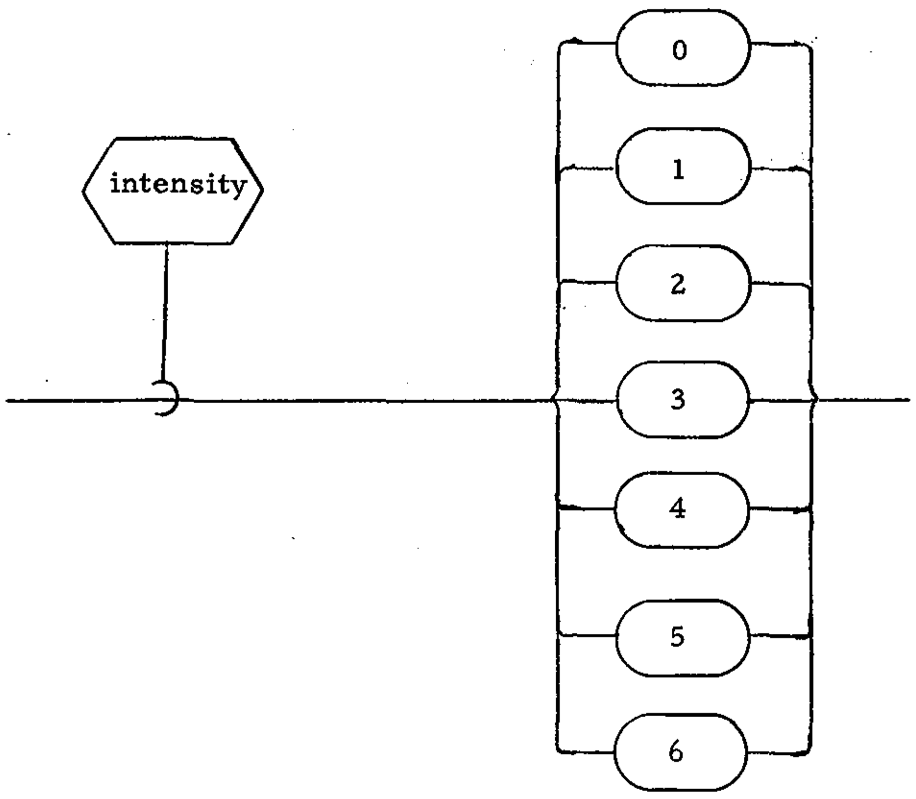
<output format> .

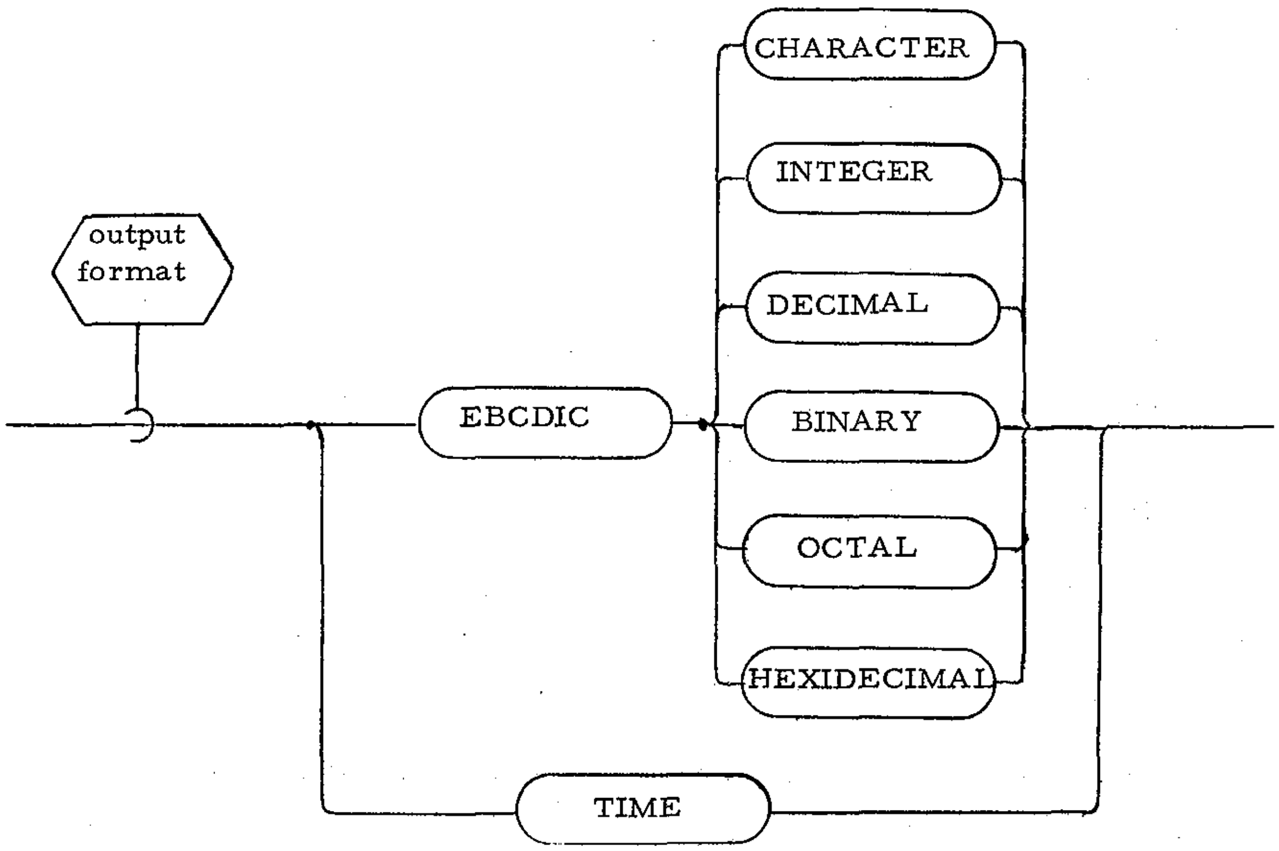
The display options are identified by the following forms:




character size
15
10
7.5
5
MM


blink data
BLINK
ON
OFF


intensity
0
1
2
3
4
5
6



The display options must be specified above; however, a term may be omitted by inserting a comma followed by the next term in the sequence.

Example:

RED,, BLINK OFF,, TIME

The display control word value list is denoted syntactically by <dcw value list>.

2.4 One-and Two-Dimensional Source Formats

In preparing HAL source text, only the single 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. The second format is used by the compiler when producing the formatted compilation listing.

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:

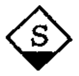
   $A^J \longrightarrow$ A**J

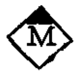
   $A^{J^K} \longrightarrow$ A**J**K

2. Operations are evaluated right to left (see Section 6.1.1).

Rules for Subscripts

1. In the syntax diagrams, the 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.

    descent to subscript line

    return from subscript line

   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.

   Example:

   $A_{K+2} \longrightarrow$ A$(K+2)

3. For certain simple forms of subscript, the parentheses may be omitted. These forms are:

o a single <number>

o a single <arith var name> (see Section 5.3).

Example:

$A_J \longrightarrow$ A$J

2.5 Comments and Blanks in the Source Text

Any HAL source text consists of sequences of HAL/SM 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 primitives are sufficient to define the boundary. In others, a blank character is required as a separator. Blanks are legal in the following situations:

- o between two primitives;
- o between two special characters; and
- o 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 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 of imbedded comments from line to line of the source text.

(BLANK)

3. HAL/SM BLOCK STRUCTURE AND ORGANIZATION

The largest syntactical unit in the HAL/SM language is the "unit of compilation." In any implementation, the HAL/SM 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 job. (A job is executable within the framework of an executive operating system, and a run time utility library.) Generally, a job contains three different types of object modules:

- o task modules - characterized by being independently executable.
- o external PROCEDURE and FUNCTION modules - characterized by being callable from other modules.
- o COMPOOL modules - forming common data pools for the program complex.

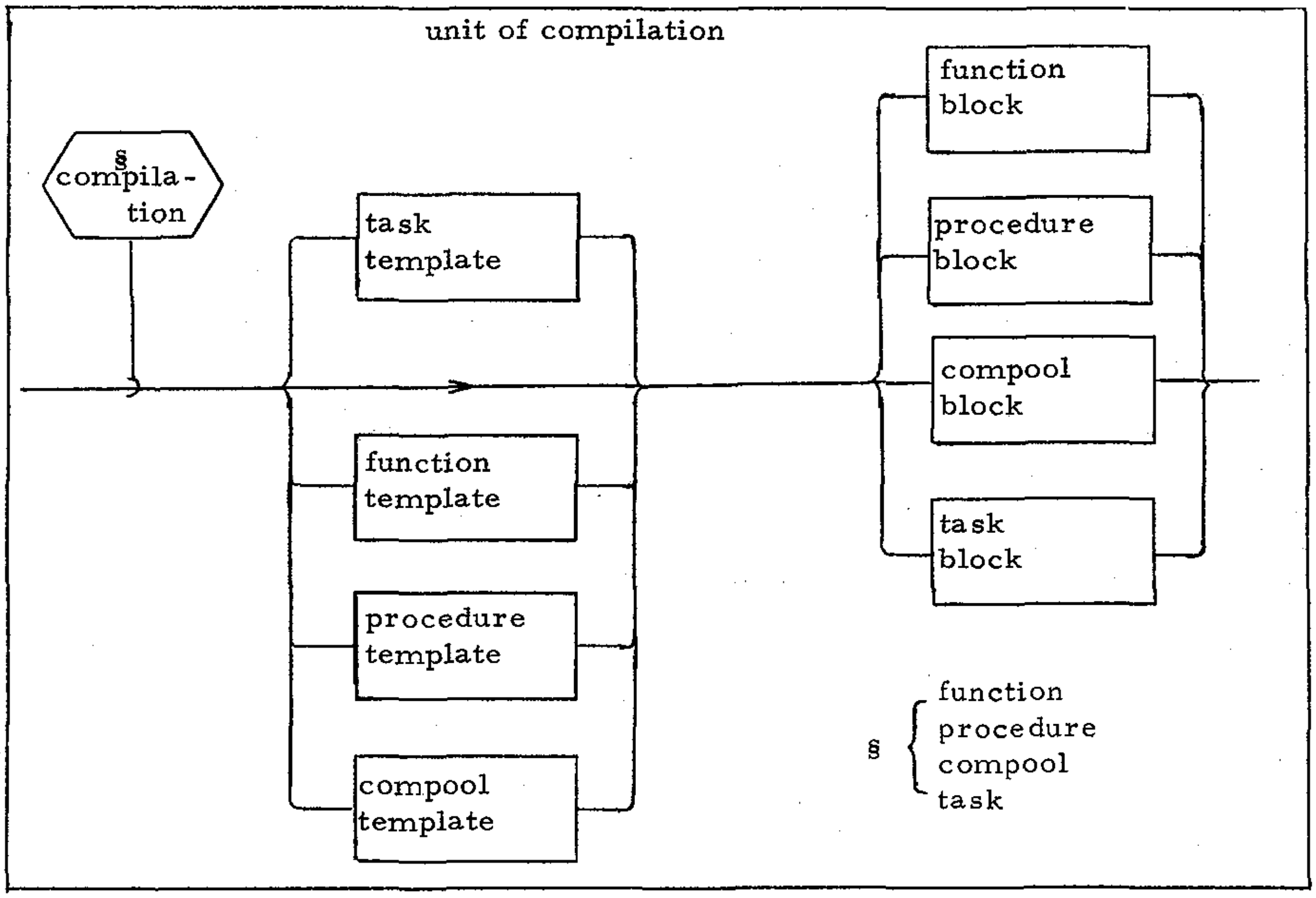
Each module originates from a unit of compilation of corresponding type.

PRECEDING PAGE BLANK NOT FILMED

3.1 The Unit of Compilation

Each unit of compilation consists of a single TASK, 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



Semantic Rules

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

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 task, 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 TASK, PROCEDURE or FUNCTION module in the program complex. The < compool block> is described in Section 3. 6.

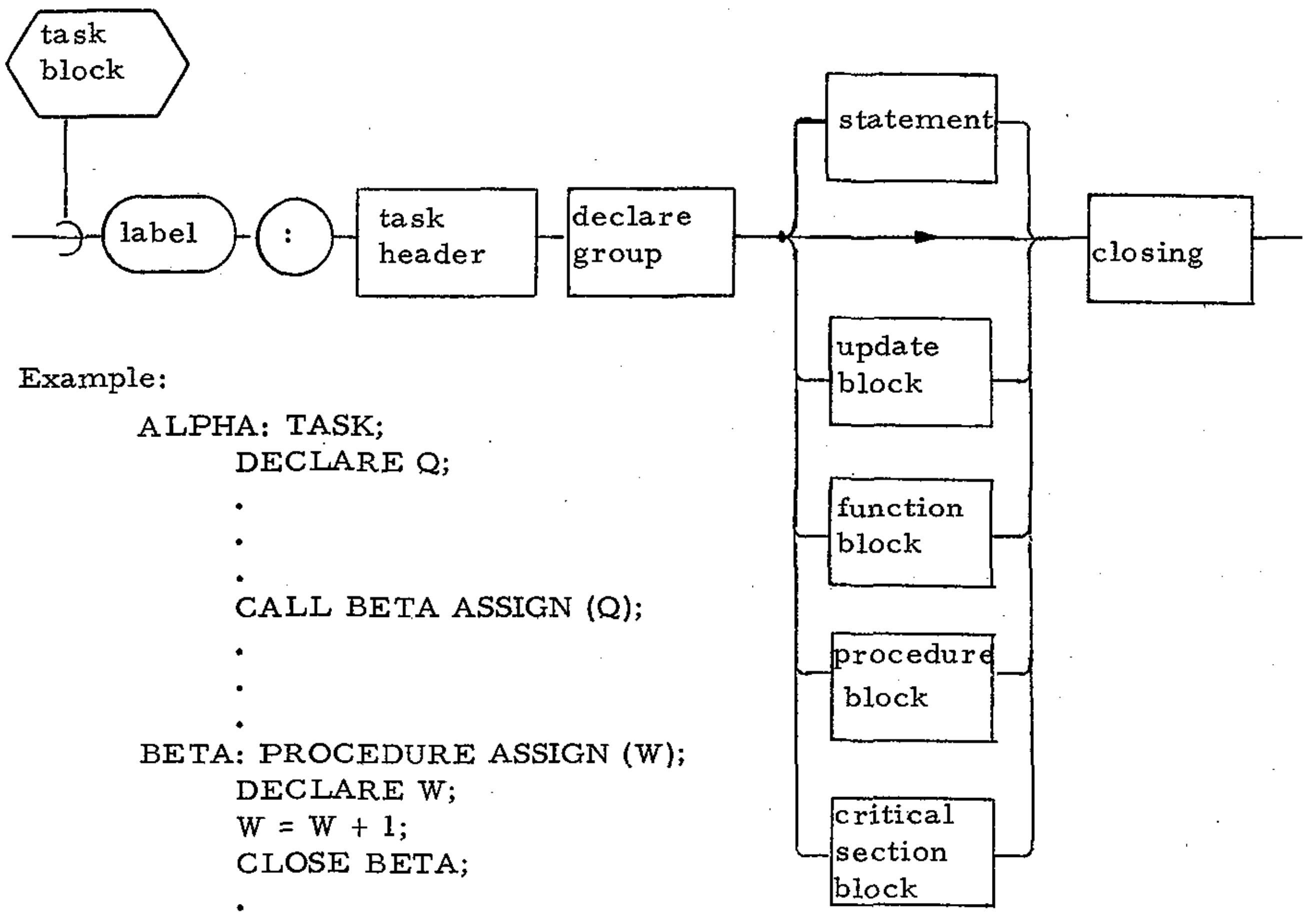
4. The code block in any < compilation > except a COMPOOL < compilation > may contain references to data in a COMPOOL < compilation > , references to other < task block > s, and invocations of external < procedure block > s or < *function* block > *s in other < compilation >* s. A < compilation > making such references must precede its code block with a block template for each such < task block > , < procedure block > , < function block > or < compool block > referenced. Block templates are described in Section 3. 7.

3.2 The TASK Block

The TASK block delimits a main, independently executable body of HAL/SM code.

Syntax

TASK block



Example:

```
ALPHA: TASK;
       DECLARE Q;
       .
       .
       .
       CALL BETA ASSIGN (Q);
       .
       .
       .
BETA: PROCEDURE ASSIGN (W);
       DECLARE W;
       W = W + 1;
       CLOSE BETA;
       .
       .
       .
       CLOSE ALPHA;
```

Semantic Rules

1. The name of the <task block> is given by the <label> prefacing the block.

2. The <task block> is delimited by a <task header> statement at the beginning, and a <closing> at the end. These two delimiting statements are described in Section 3.8.1 and 3.8.4, respectively.

3. The contents of a < task block> consist of a < declare group> used to define data local to the < task 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-defining way.

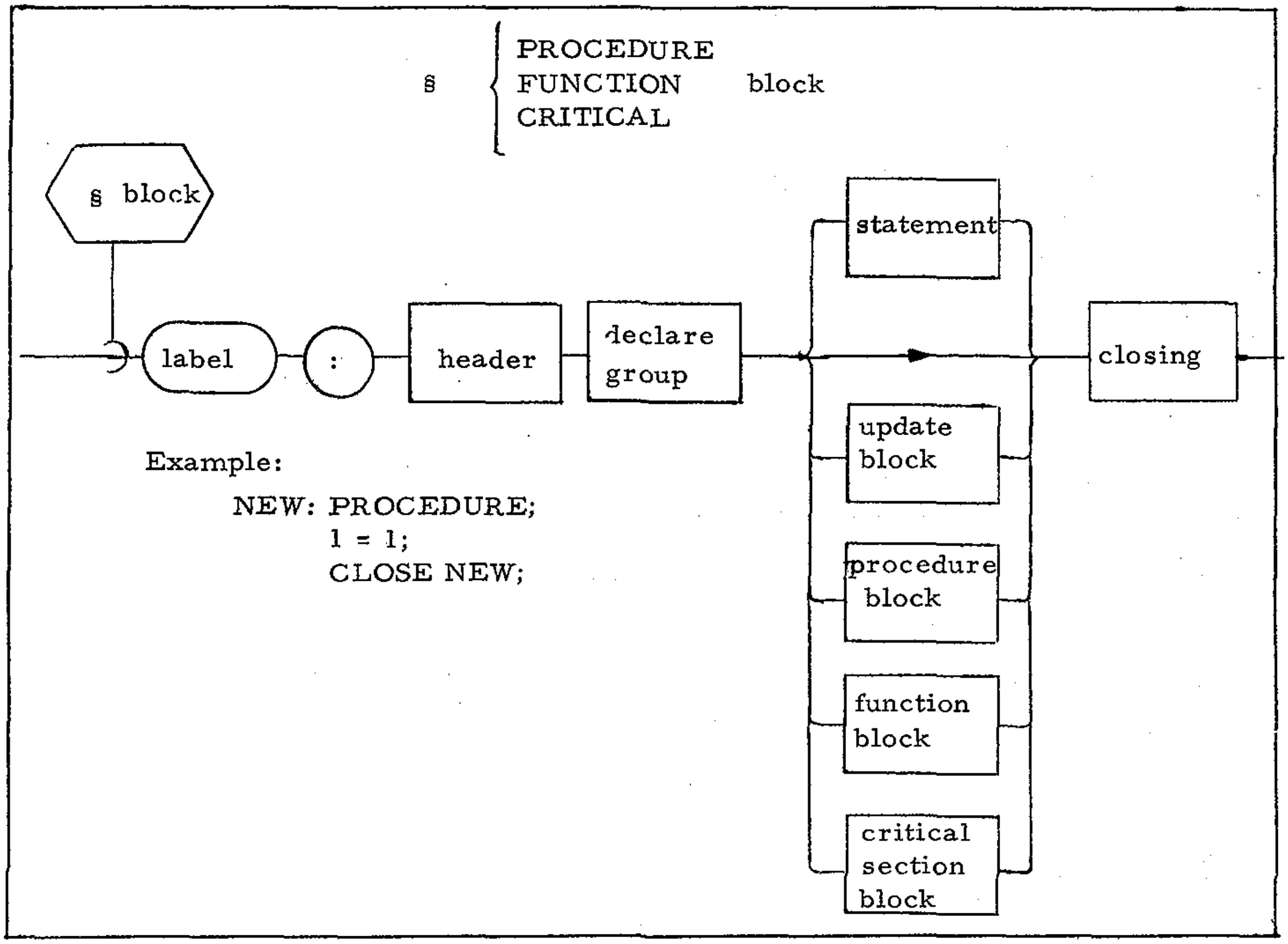
5. PROCEDURE, FUNCTION, CRITICAL SECTION, and UPDATE blocks may appear nested within a < task block> . The blocks may be interspersed between the < statement> s of the < task block> , and with the exception of the UPDATE and CRITICAL SECTION blocks, are not executed in-line.

6. Execution of a < task block> is accomplished by scheduling it as a task under the control of MOSS (see Section 8).

3.3 PROCEDURE and FUNCTION Blocks

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

Syntax



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 Section 3.8.1 through 3.8.4.

3. The contents of the block consist 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> s may modify this normal sequencing in a well-defined way.

5. The block may contain further nested PROCEDURE, FUNCTION, CRITICAL SECTION, and UPDATE blocks. The nested blocks may appear interspersed between the <statement> s of the outer block and, except for the UPDATE and CRITICAL SECTION blocks 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.

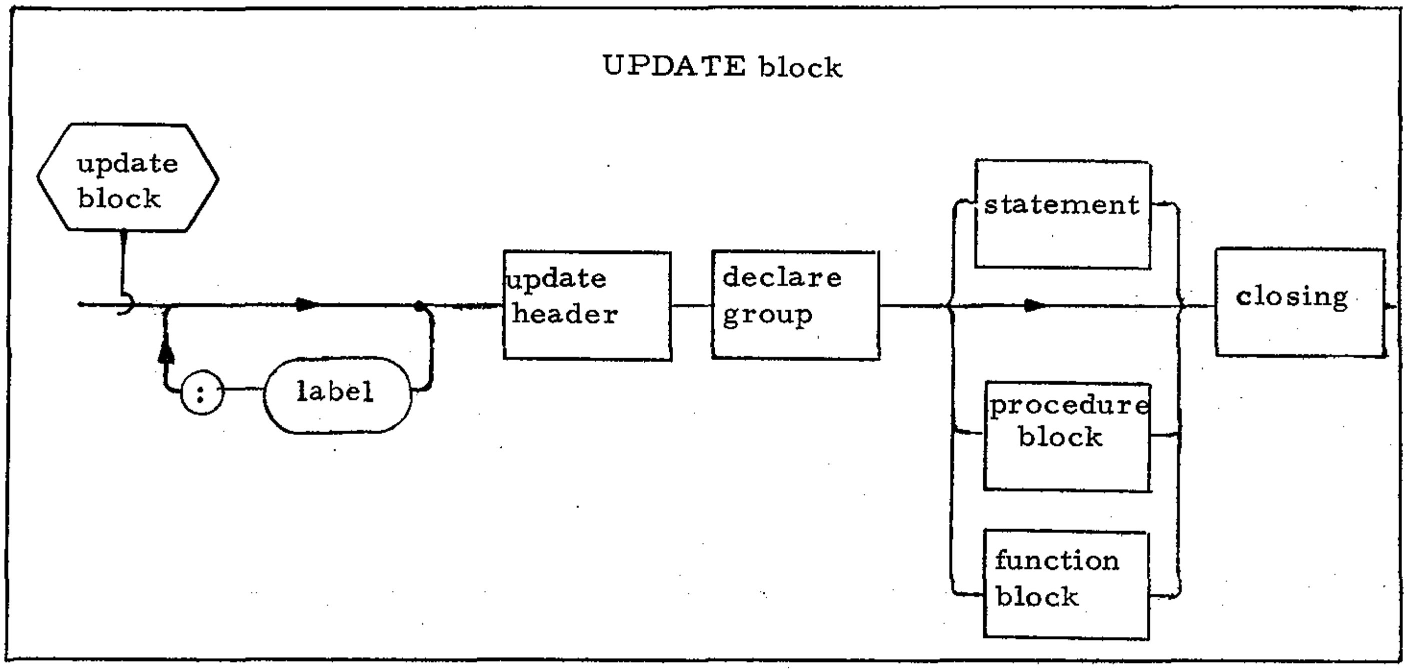
6. Execution of a < procedure block> is invoked by the CALL statement (see Section 7.4). Execution of a < function block> is invoked by the appearance of its name in an expression (see Section 6.4).

7. 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



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 3.8.1 and 3.8.4.

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 <updated 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:

   o I/O statements (see Section 10);

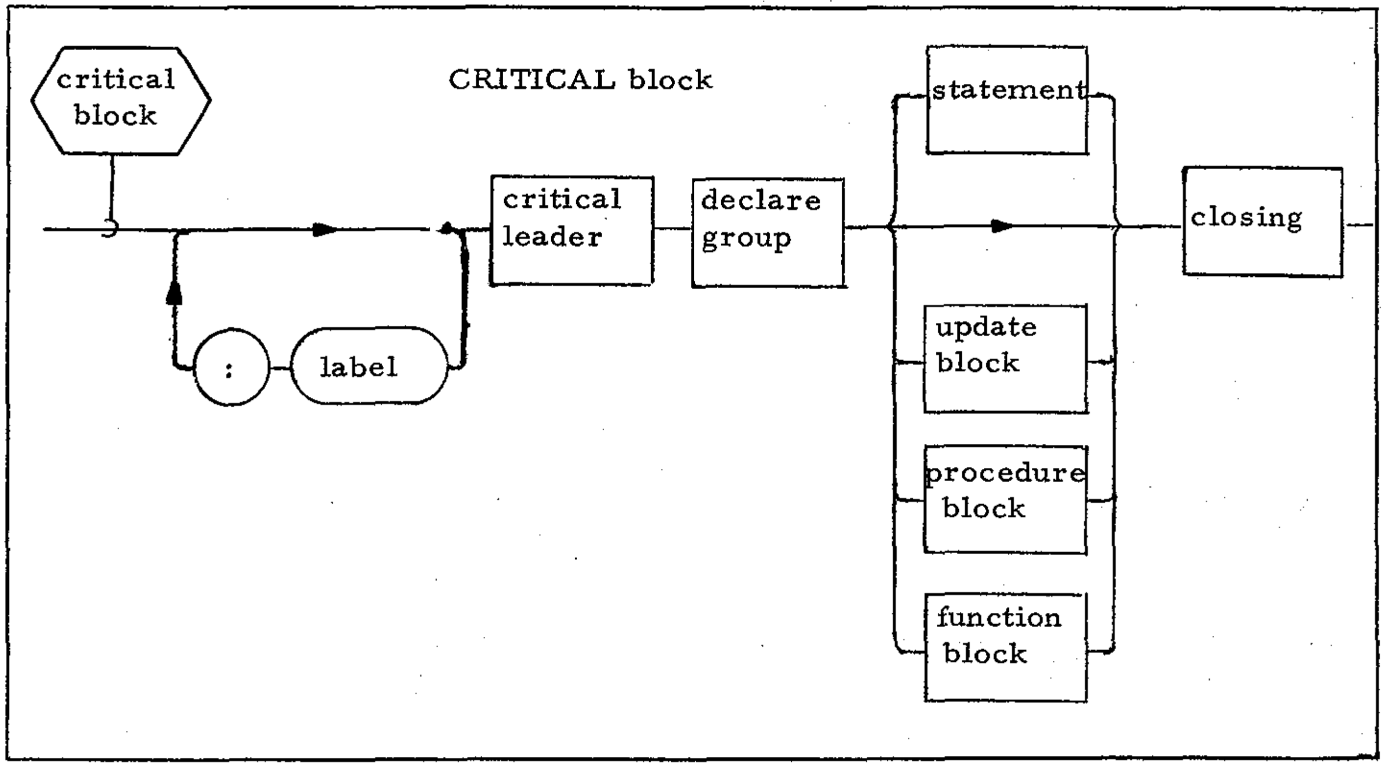
   o invocations of <procedure block> s or < function block> s not themselves nested within the < update block> ; and

   o real time programming statements.

3.5 <u>The CRITICAL SECTION Block</u>

The CRITICAL block is used to define the beginning and end of a CRITICAL SECTION (that section of a task for which the MOSS critical processing mode must be invoked).

<u>Syntax</u>



<u>Semantic Rules</u>

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

2. The block is delimited at its beginning by a <critical header> statement, and at the end by a <closing>. The delimiting statements are described in Sections 3.8.1 and 3.8.4.

3. The contents of the block consist of a <declare group> used to declare data local to the <critical 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.

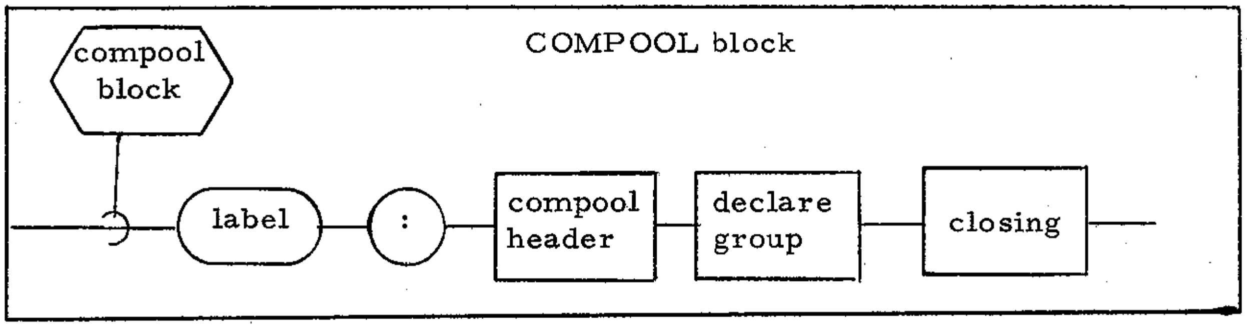
5. The block may contain further nested PROCEDURE, FUNCTION, and UPDATE blocks. The nested blocks may appear interspersed between the <statement> s of the outer block and except for the UPDATE block are not executed in-line. A consequence of this rule is that PROCEDURE and FUNCTION blocks may be nested within each other to an arbitrary depth.

6. A <critical block> is treated like a <statement> in that it is executed in-line. In this respect it is different from other code blocks except for the <update block> .

3.6 The COMPOOL Block

The COMPOOL block specifies data in a common data pool to be shared at run time by a number of TASK, PROCEDURE or FUNCTION modules.

Syntax



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 3.8.1 and 3.8.4.

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.

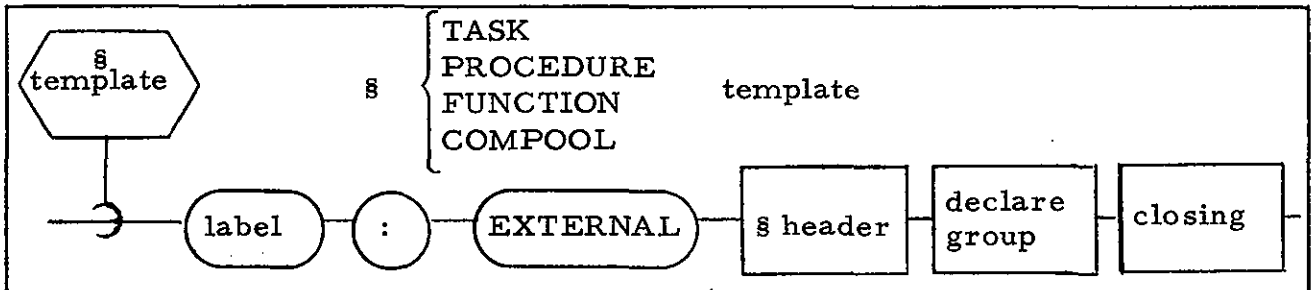
4. The maximum number of <compool block> s existing in a program complex is implementation dependent.

3.7 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 TASK, 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, TASK, PROCEDURE, FUNCTION, and COMPOOL templates, all being syntactically similar (see Section 3.1).

Syntax



Example:

```
        ETA:    EXTERNAL COMPOOL;
                DECLARE S SCALAR;
                CLOSE ETA;
```

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 3.8.1 through 3.8.4.

3. The contents of the block template consist only of a <declare group>, which has the following significance:

   o in a <task template>, the <declare group> contains no statements. All information about external programs is contained in the <task header>.

   o 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 3.8.2 and 3.8.3).

4. The keyword EXTERNAL preceding the header statement of the block templete distinguishes it from an otherwise identical code block. To a HAL/SM 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.

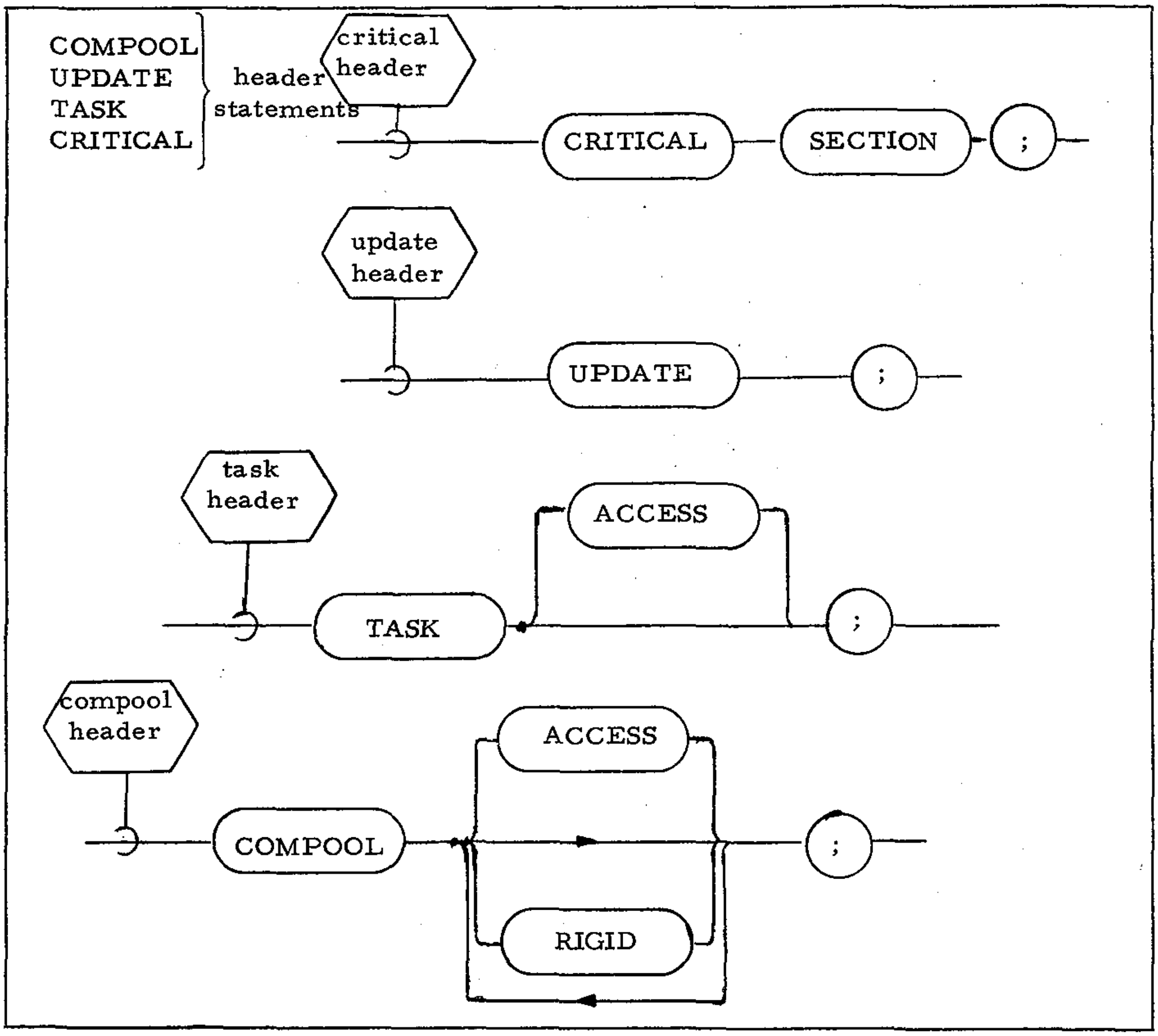
3.8 Block Delimiting Statements

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 on entry. A COMPOOL block, containing only declarations of data, is, of course, not executable at all.

3.8.1 *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, TASK, CRITICAL SECTION and UPDATE header statements.

Syntax



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, TASK, CRITICAL SECTION, and UPDATE.

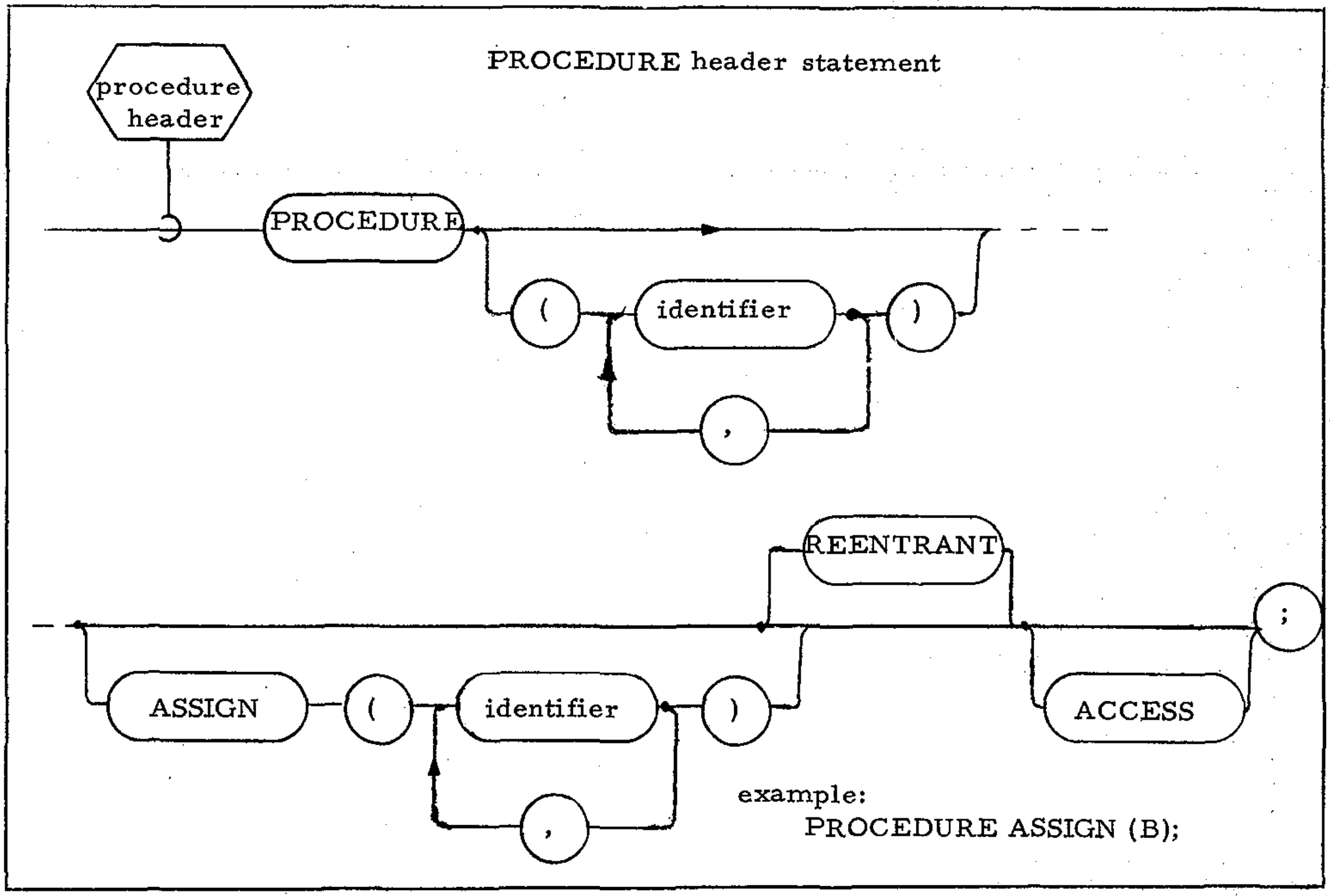
2. The keyword ACCESS causes managerial restrictions to be placed upon the usage of the block in question. The manner of enforcement of the restriction is implementation dependent.

3. The keyword RIGID causes COMPOOL data to be organized in the order declared and not rearranged by the compiler.

3.8.2 The PROCEDURE Header Statement

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

Syntax



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 does not specify the keyword REENTRANT, 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 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:

   o 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.

   o 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.

   o 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:

- o UPDATE blocks*, CRITICAL SECTIONs, and inline functions within a REENTRANT block may not declare any local data, STATIC or AUTOMATIC.

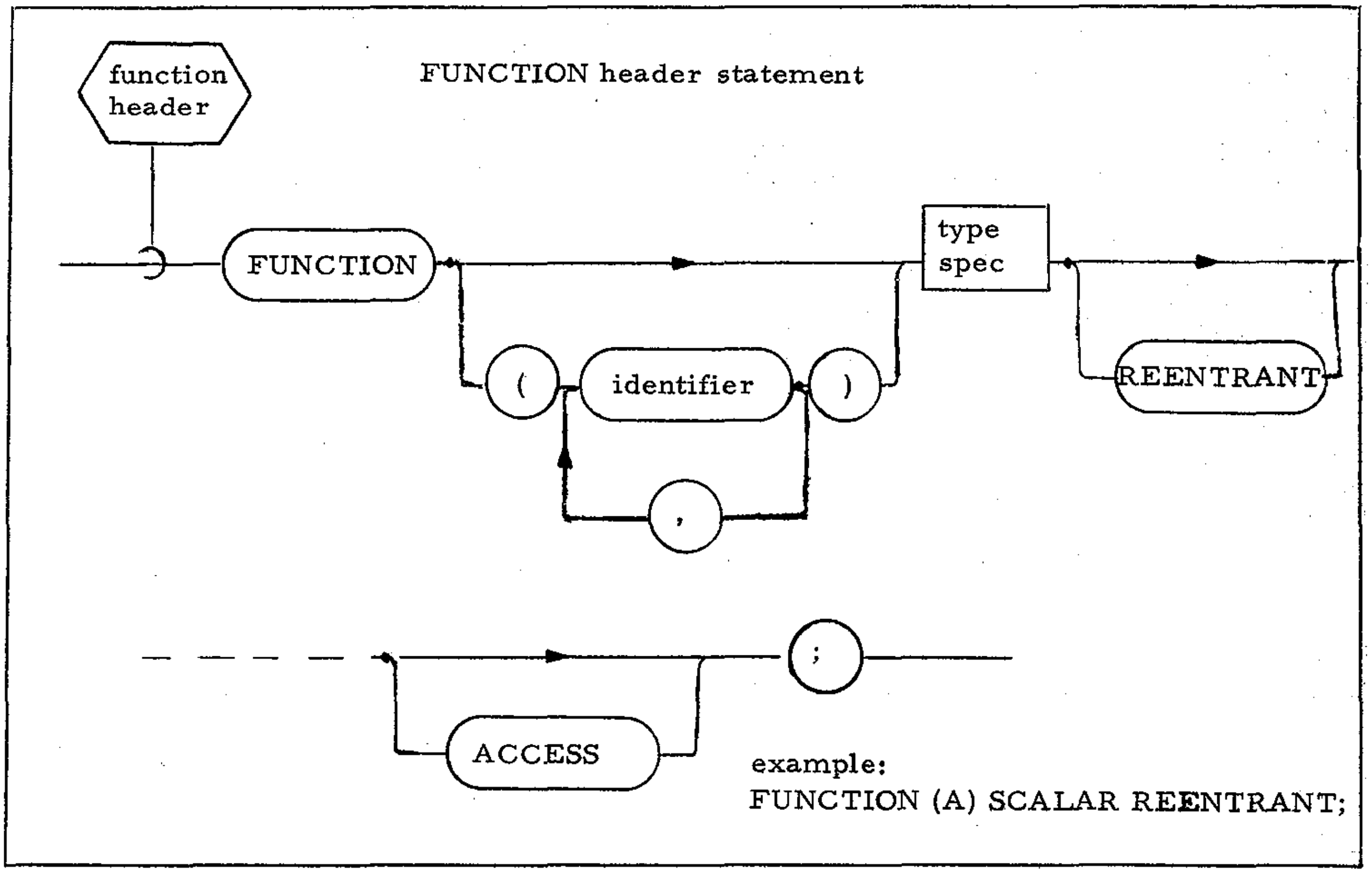
- o A procedure or function called by a REENTRANT block must itself also be REENTRANT.

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.

3.8.3 The FUNCTION Header Statement

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

Syntax



*Any use of UPDATE blocks and LOCK data should be carefully analyzed with respect to unfavorable interactions with REENTRANT blocks.

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 6.4).

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 <function template>.

4. A <type spec> identifies the type of the <function block> or <function template>. A <function block> may be of any type except event. A formal description of the type specification given by <type spec> is given in Section 4.7.

5. If the <function header> statement does not specify the keyword REENTRANT, 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 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:

   o 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.

   o 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.

   o 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:

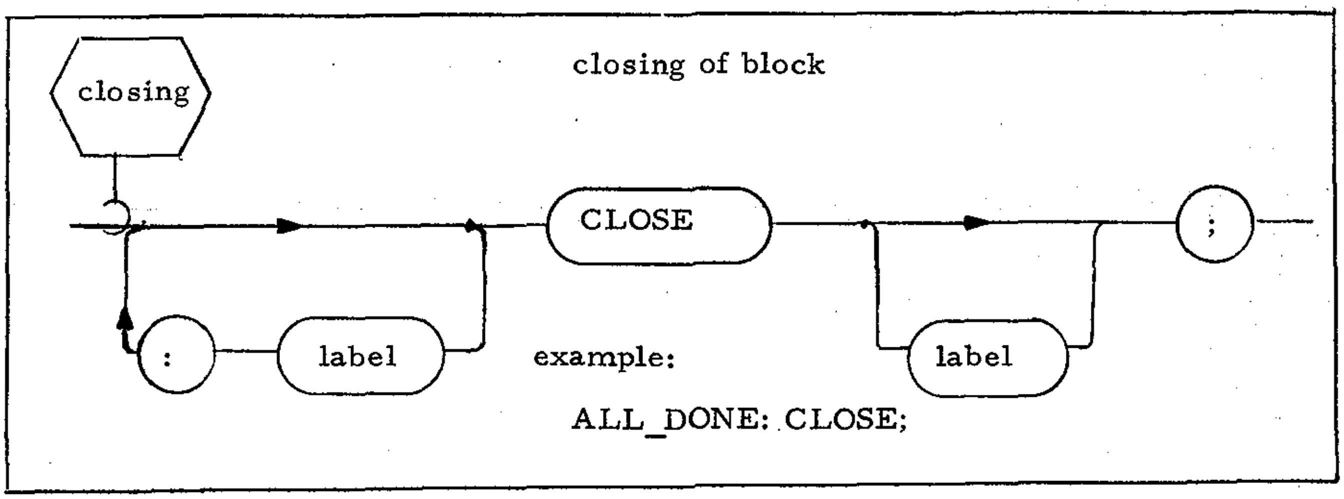
- o UPDATE blocks*, CRITICAL SECTIONs, and inline functions within a REENTRANT block may not declare any local data, STATIC or AUTOMATIC.
- o A procedure or function called by a REENTRANT block must itself also be REENTRANT.

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.

3.8.4 The CLOSE Statement

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

Syntax



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 TASK, PROCEDURE, CRITICAL 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 TASK, PROCEDURE, FUNCTION, CRITICAL, or UPDATE block may be labeled as if it were a <statement>. The <closing> s of COMPOOL blocks and block templates cannot be labeled.

* Any use of UPDATE blocks and LOCK data should be carefully analyzed with respect to unfavorable interactions with REENTRANT blocks.

3.9 Name-Scope Rules

By using the code blocks described, and by taking advantage of their nesting property, the modularization of HAL/SM <compilation> s may be effected. 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 the block including all blocks nested within it.

2. A 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 label 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.

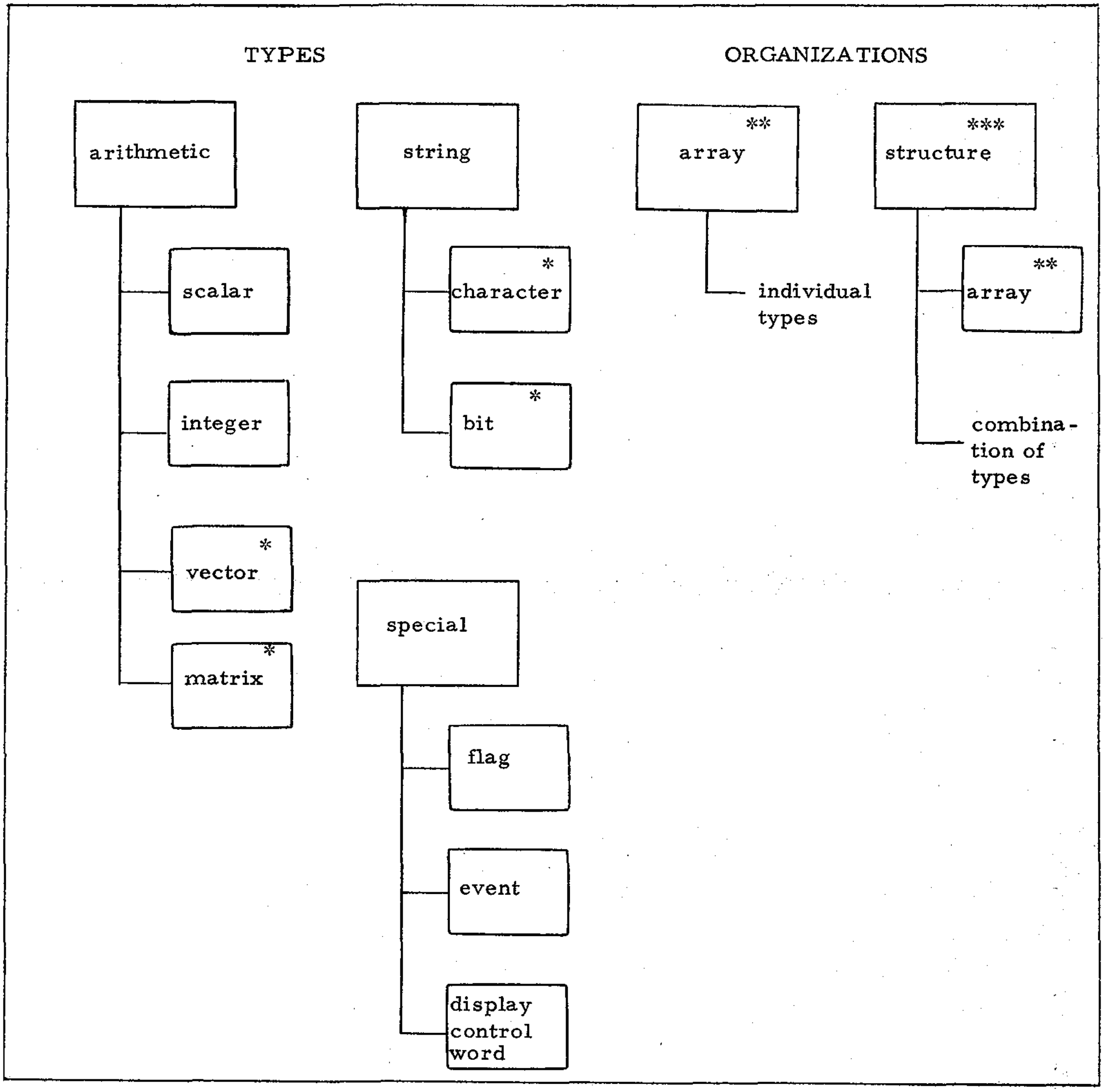
Example:

```
            ALPHA: TASK;
                   DECLARE X;  <------  X known everywhere
outer              DECLARE Y;           this Y known everywhere except in BETA.
name
scope                  .
                       .
                       .
                       .
            BETA: PROCEDURE;  <------ BETA is known everywhere;
                   DECLARE Y;  <------  new Y known in BETA only
inner              DECLARE Z;           Z known in BETA only
name
scope                  .
                       .
                       .
                       .
                   CLOSE BETA;
                       .
                       .
                       .
            DELTA: Y = 0;     <------   DELTA not known in BETA
                       .
                       .
                       .
                       .
                   CLOSE ALPHA;
```

4. DATA AND OTHER DECLARATIONS

The HAL/SM 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 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.

HAL DATA TYPES AND ORGANIZATIONS



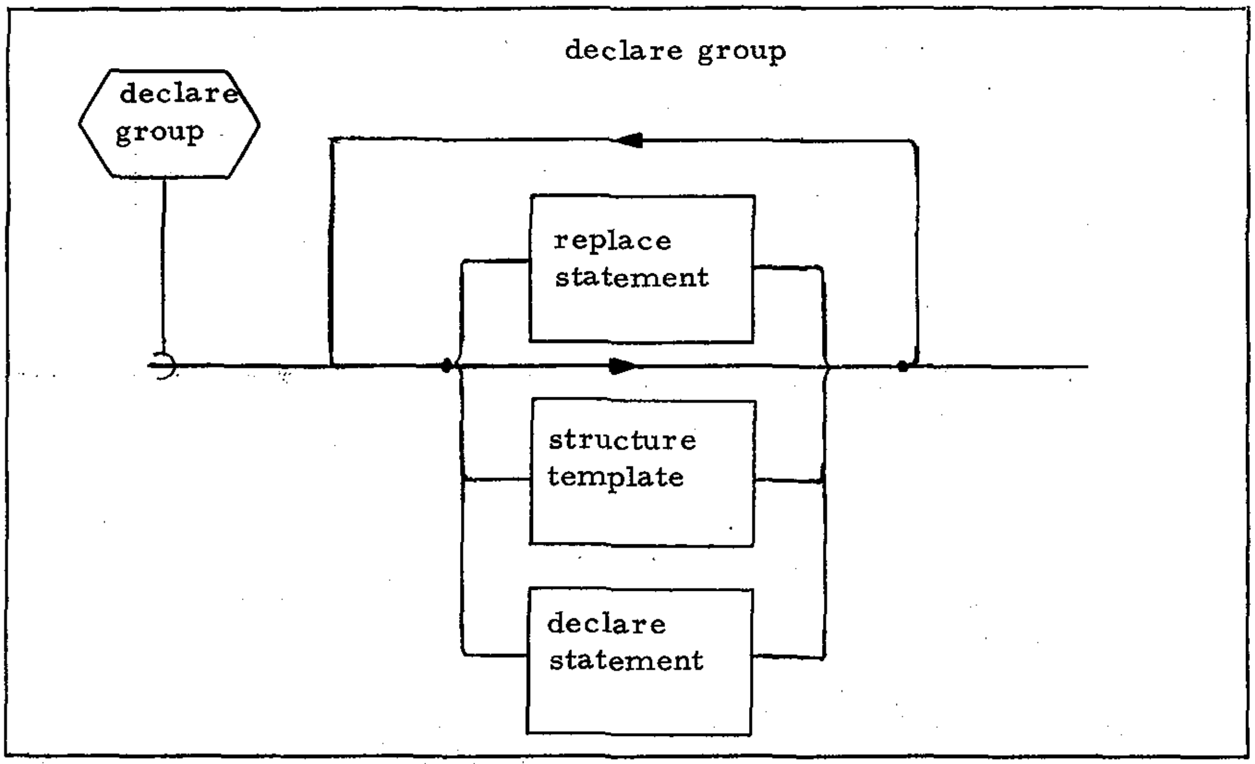
* Component Subscripting (see Section 5. 3. 5) Allowed.
** Array Subscripting Allowed.
*** Structure Subscripting Allowed.

Figure 4-1

4.1 The Declare Group

A <delcare 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



Semantic Rules

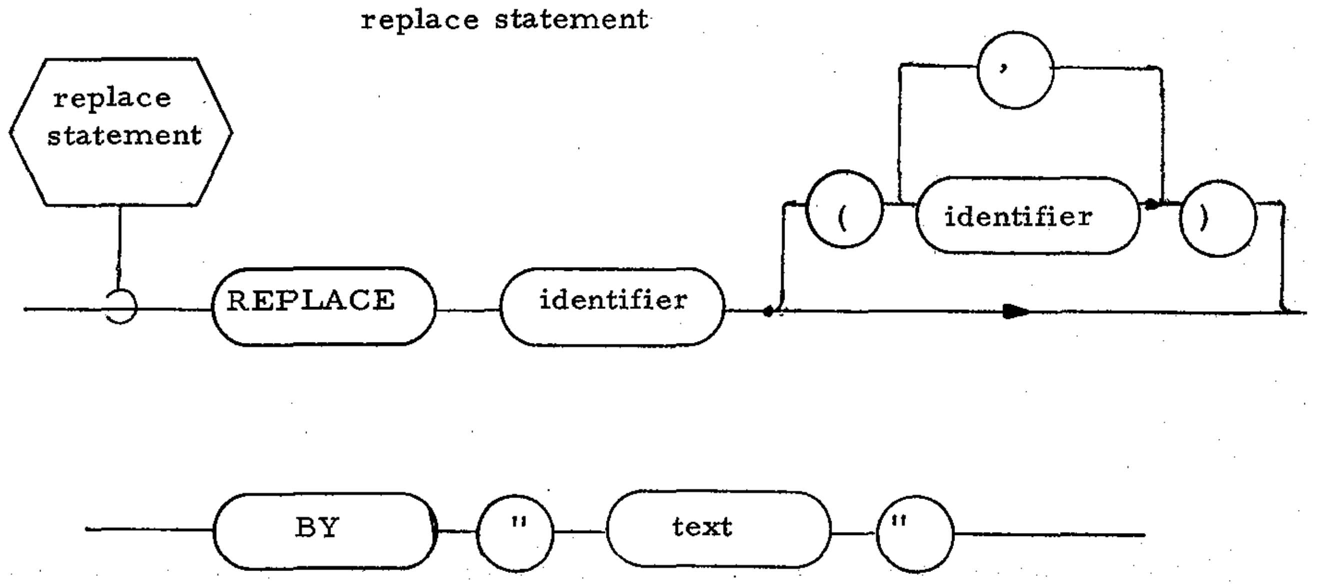
1. A <declare group> may simply be empty, or it may contain <replace statement> s, <structure template> s, and <declare statement> s. The form of each of these constructs is defined in this section.

2. The name-scope (see Section 3.9) 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



examples:

```
REPLACE ALPHA BY "J+1";
REPLACE BETA (X,ANGLE) BY "SIN (X ANGLE) - EXP (X)/X";
```

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 <replace 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/SM characters except instances of an unpaired double quote (") character. A double quote character (") is indicated within <text> by two such characters in succession ("").

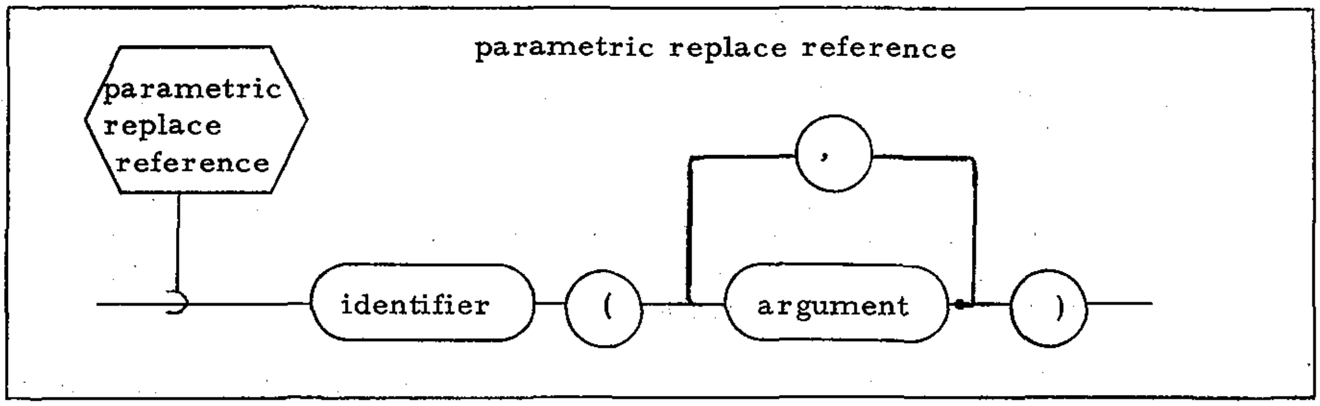
Semantic Rules: Parametric Replacements

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/SM 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/SM 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



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 definition's <text> . Note that if the parameter in question does not occur within the source macro definition < 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.

   o 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.

   o The <text> string comprising an < argument> may be empty. The value substituted in such a case is a null string.

   o 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> .

   o 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> .

- o 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.

- o Commas are not separators between <argument>s under the following circumstances:

  - within a character literal,

  - within REPLACE<text>, or

  - 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 argument with the characters surrounding the ¢-enclosed parameter (a blank is considered as a character).

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;
```

Enclosing the entire reference within ¢ 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/SM, 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 root is connected to 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.

Figure 4-2 illustrates a typical tree organization.

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 Figure 4-2.

4. The tree organizations of two templates are considered to be equivalent for the purposes of various HAL/SM 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 not identical.

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

TREE DIAGRAM FOR A TYPICAL STRUCTURE
TEMPLATE

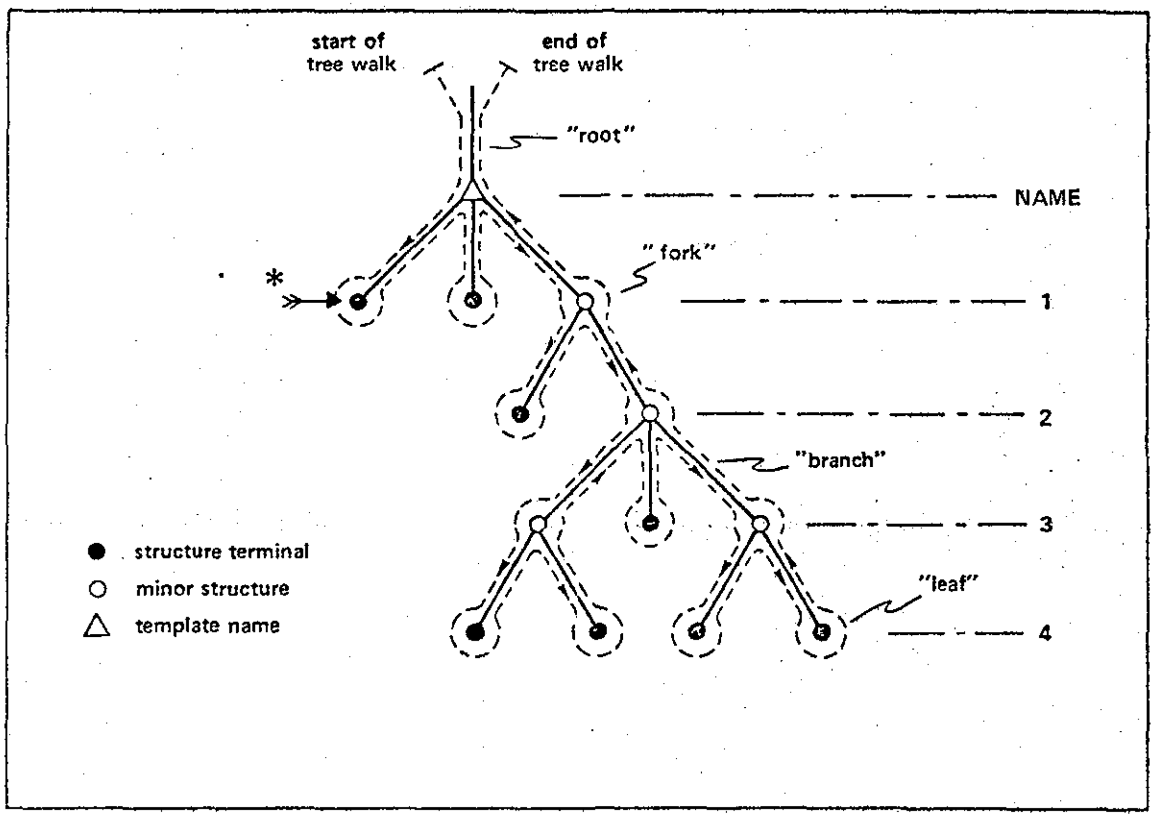
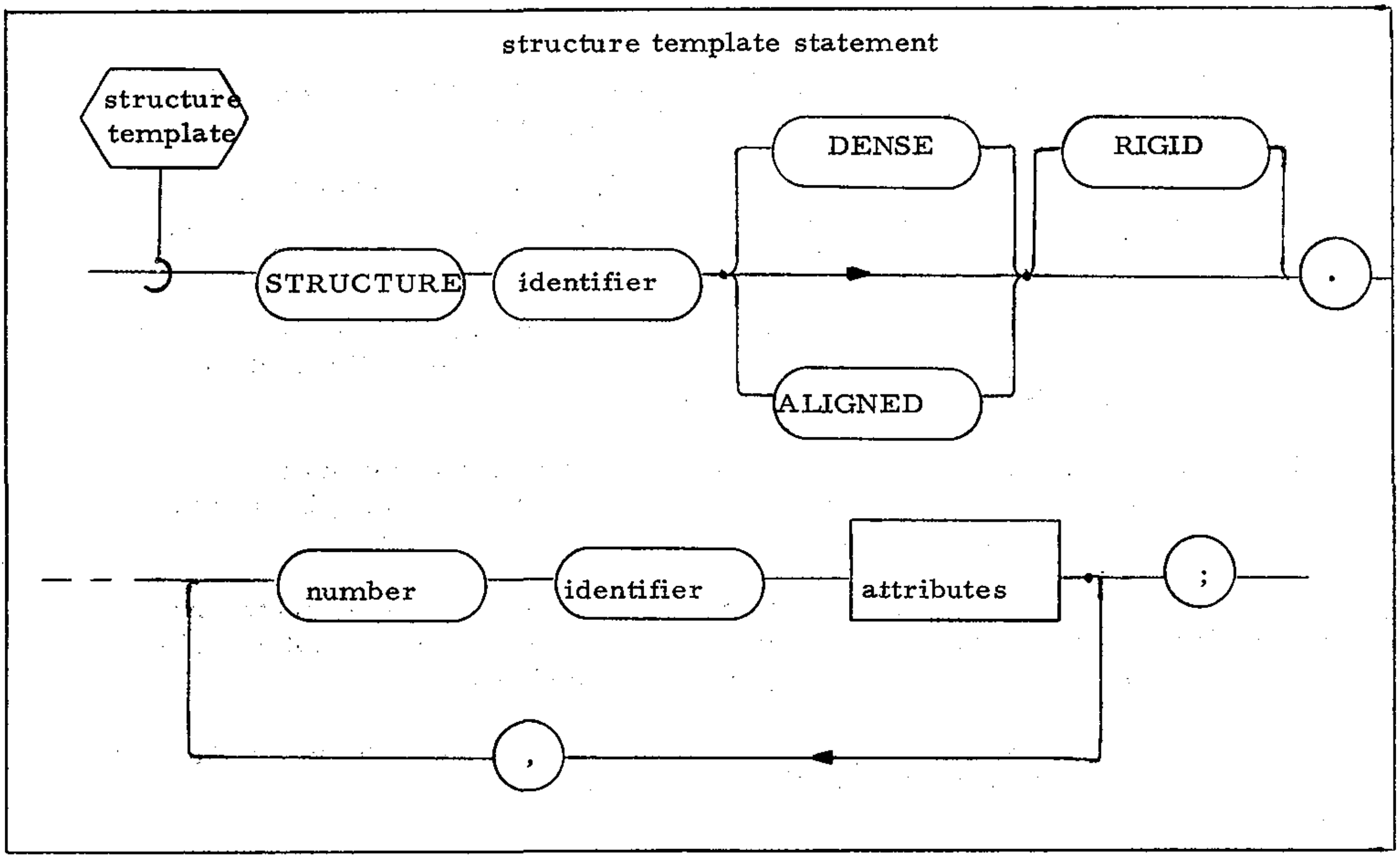


Figure 4-2

Syntax



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 default 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 organized 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.

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 or structure terminal is separated from the next by a comma.

Name Uniqueness Rules

1. The <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:

   o Any <identifier> used for a node in one template may duplicate an <identifier> used for a node in another template.

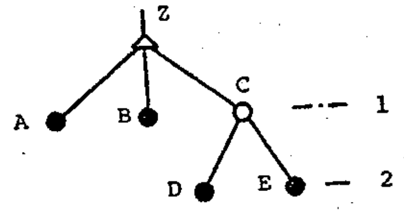
   o Any <identifier> used for a node in a given template may duplicate another <identifier> used for a different node in the same template, provided that a qualified reference can distinguish the two nodes.

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:

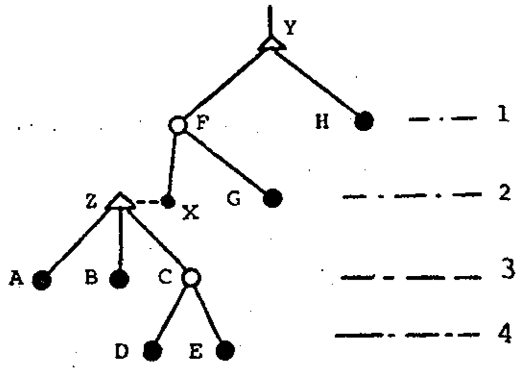
A. definition of a template Z

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



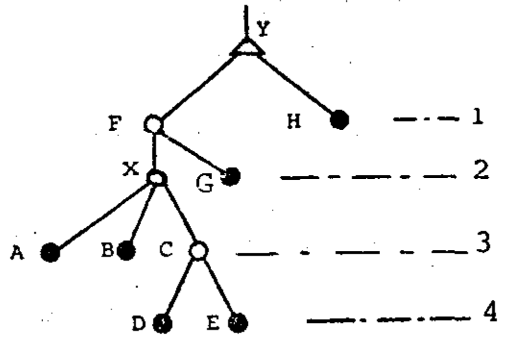
B. definition of a template Y with Z nested within it

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



C. equivalent form of template Y without nesting

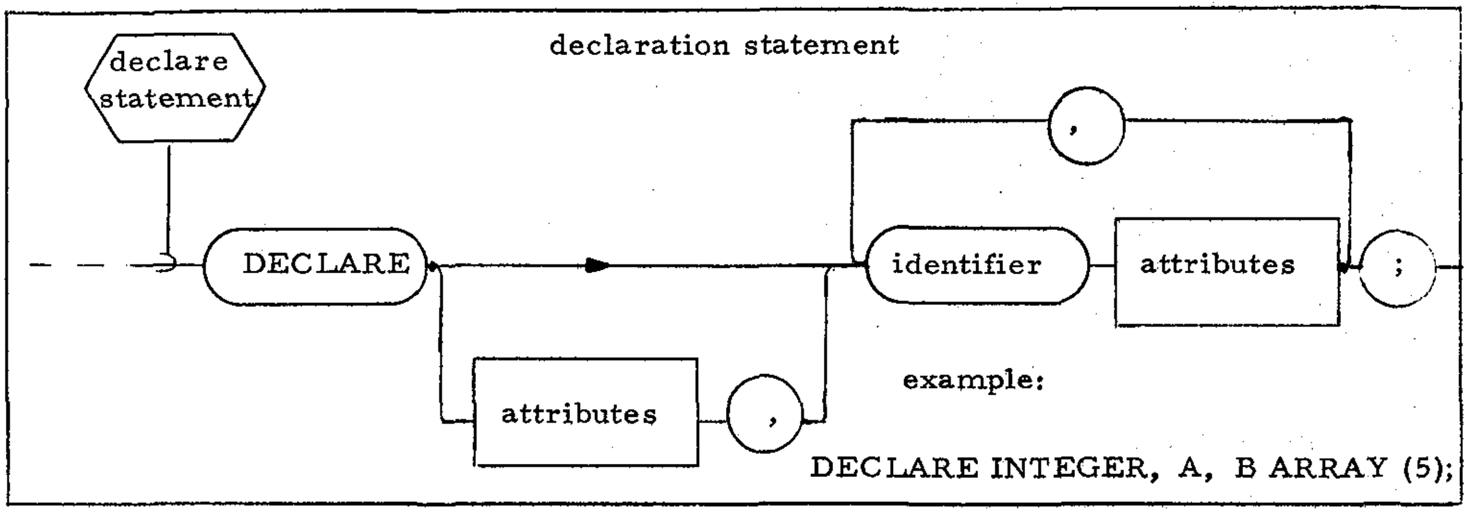
```
STRUCTURE Y;
        1 F,
           2 X,
              3 A SCALAR,
              3 B VECTOR(4),
              3 C,
              4 D MATRIX(4, 4),
              4 E BIT(3),
           2 G INTEGER,
        1 H CHARACTER(10);
```



4.4 The DECLARE Statement

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

Syntax



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> .

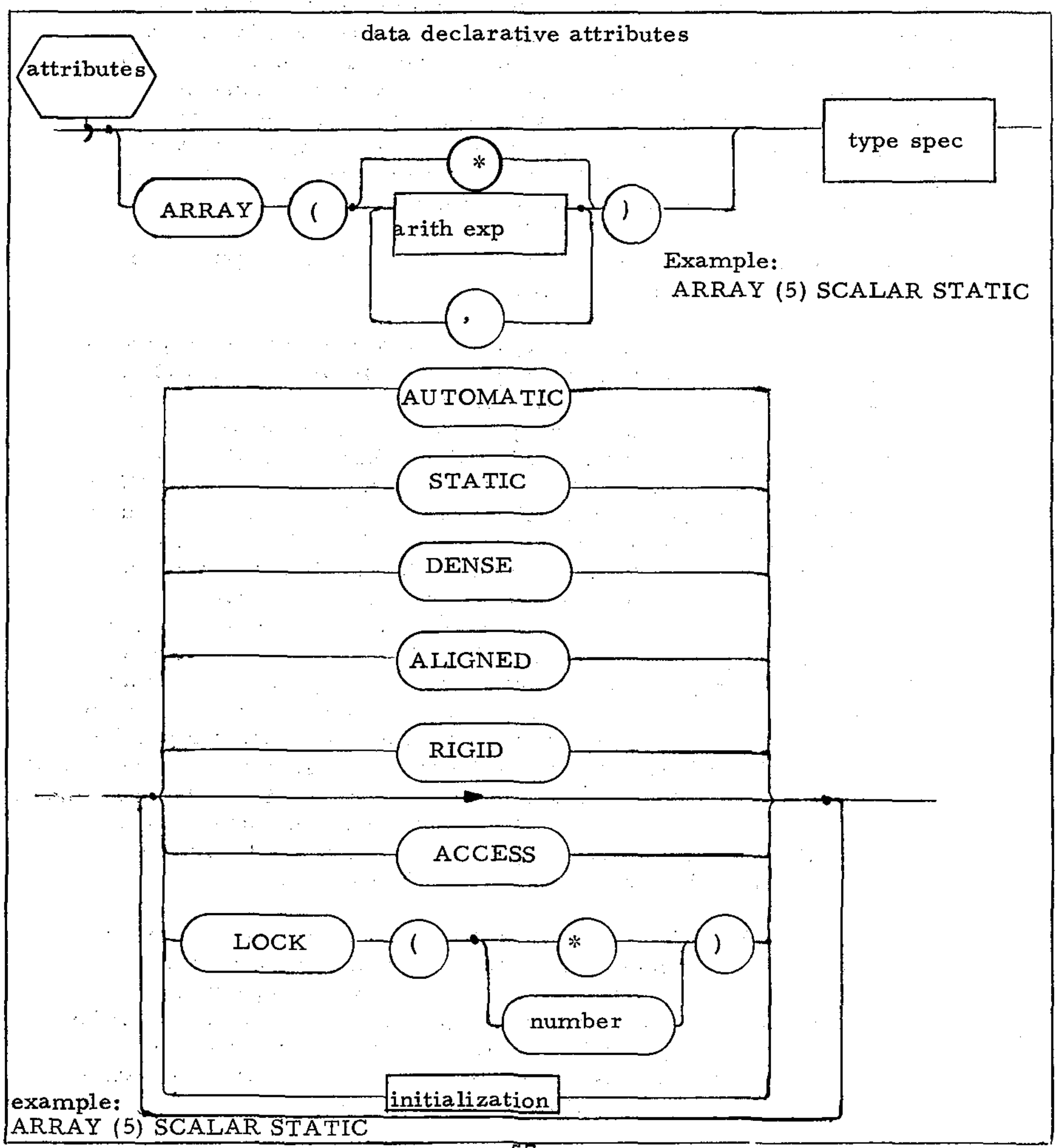
3. The name-scope of any of the <identifier> s defined in a < declare statement> is the code block containing the <declare group> of which the <declare statement> is a part (see Section 3.9). In any name-scope all such <identifiers> must be unique.

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> appears in a< declare statement> , the <identifier> defined is a "simple variable," or a "major structure" with predefined template. If < attributes> appears 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



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 of scalar expression computable at compile time. The value is rounded to the nearest integer, and indicates the number of elements in a dimension. Its value must lie between 2 and an implementation-dependent maximum. The maximum value of N is implementation dependent. A single asterisk denotes a linear array, the number of elements of which is unknown at compile time.

3. Following the < type spec > a number of minor attributes applicable to the < identifier > can appear. These are:

   o 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 (except in reentrant procedures and functions where these keywords refer to both allocation and initialization). 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. In the case of reentrant procedures and functions, the STATIC attribute implies compile time allocation and static initialization, and the AUTOMATIC attribute implies dynamic allocation and initialization upon each entry to the block. If neither keyword appears, then STATIC is assumed.

   o 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. The <number> indicates the "lock group" of the <identifier> and lies between 1 and an implementation-dependent maximum. A "*" indicates the set of all lock groups. The purpose of the attribute is described in Section 8.14.

- <initialization> - this attribute describes the manner in which the values of an <identifier> are to be initialized. It is described in Section 4.8.

- RIGID - Although legal on 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> or <program block>, or to the assign parameters of procedure blocks.

4. The attributes DENSE, ALIGNED, and RIGID are illegal for major structures.

5. The <initialization> attribute may not be applied to formal parameters of procedures and functions or any <identifier> of EVENT or FLAG type.

Restrictions for Structure Terminals

1. The asterisk form of array specification is not allowed.

2. The < identifier> may not be defined to be a major 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.

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 <type spec>.

   Example:

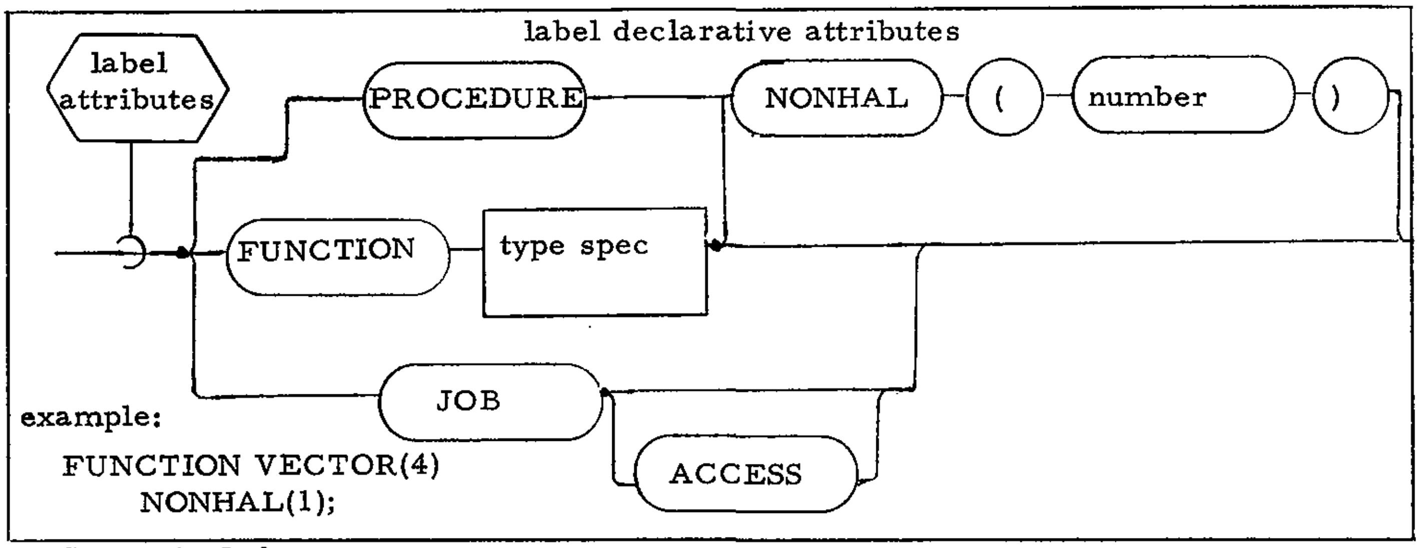
   ```
   STRUCTURE Y:
      1 A SCALAR,
      1 B VECTOR(4),
      1 D MATRIX(4,4);

   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.

4.6 Label Declarative Attributes

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

Syntax



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:

   o they must have at least one formal parameter, and

   o 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 or FLAG.

2. The NONHAL (< number >) indicates that an external routine written in some other language is being declared. 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.

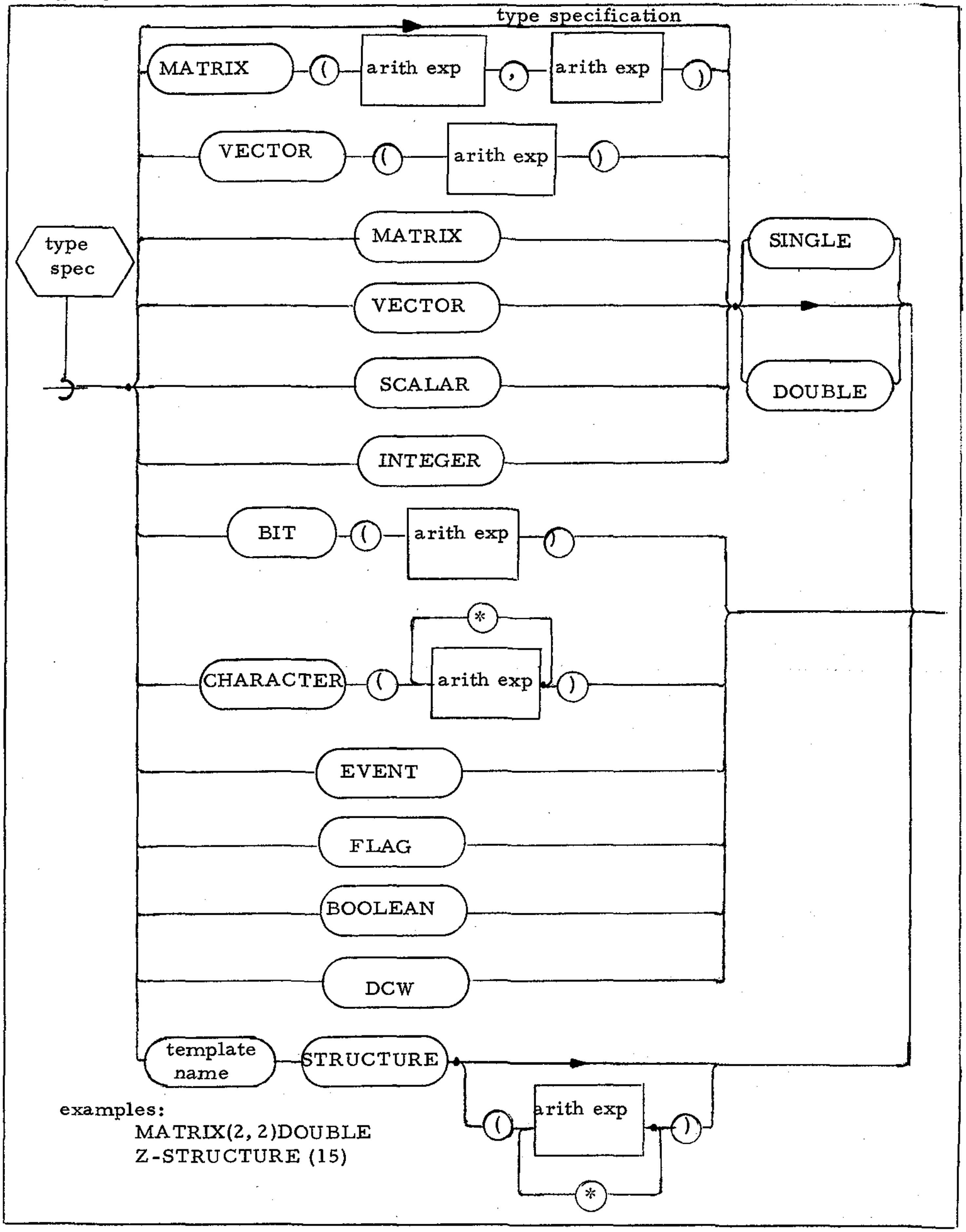
3. The form JOB is used to define the name of a job. The < identifier > being defined must be limited to eight characters or less and must be identical to the MOSS Job ID specified in JCL. This form may only be specified in program level declarations.

4. The keyword ACCESS may be attached to the JOB label< attribute> . It denotes that managerial restrictions are to be placed on which < compilation> s may reference the job identifier. The manner of enforcement is implementation dependent.

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



examples:
MATRIX(2, 2)DOUBLE
Z-STRUCTURE (15)

General Semantic Rules

1. If <type spec> is empty (i.e., there is no specification present) then the interpretation is as follows:

   o If the <type spec> is that of a simple variable or structure terminal, then the implied type is SCALAR with SINGLE precision.

   o The <type spec> is otherwise 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. Its value is rounded to the nearest integer.

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. Matrix type is indicated by the keyword MATRIX. 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, a 3-by-3 matrix is implied.

2. Vector type is indicated by the keyword VECTOR. If present, the parenthesized <arith exp> indicates the length of the vector. In the absence of a length specification, a 3-vector 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 and Boolean and 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.

Rules for Structure Type

1. The conditions for the < type spec > indicating a minor structure are 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.

5. If the < identifier > name defined 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. 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.

Rules of Event and Flag Types

1. 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 may not be used as an input format parameter, nor may it be a structure terminal.

2. The keyword FLAG indicates a flag type, which is used for real time programming (see Section 8). Flag types are not actual variables and may only be used as described in Section 8.

3. An <identifier> of flag types may only be declared in the declare group in a COMPOOL block and may not be arrayed, a structure terminal, nor used as a formal parameter.

Rules for DCW Types

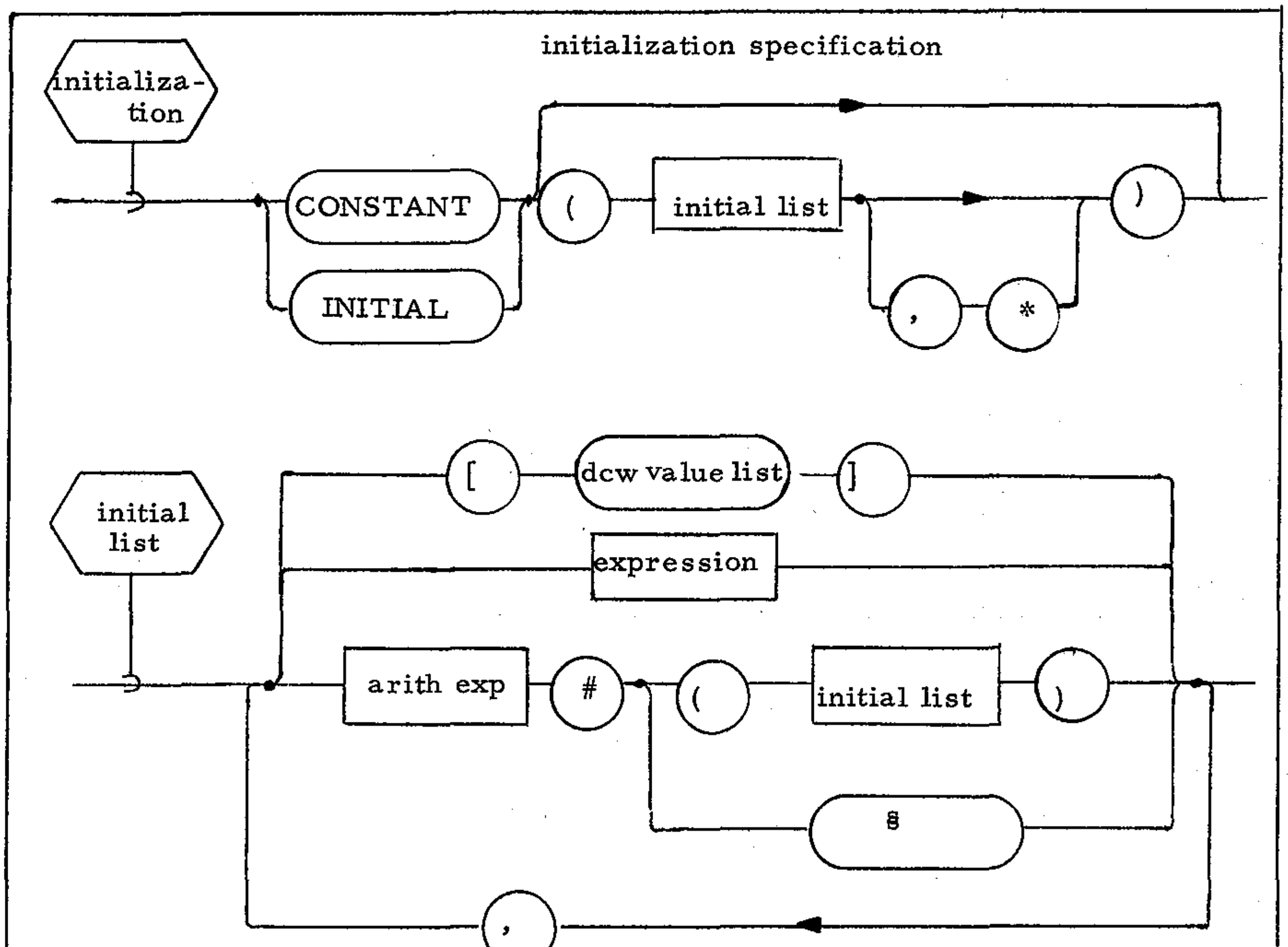
1. The keyword DCW indicates a display control word type, which is used for Control and Display console communications.

2. An <identifier> of DCW type is used to specify the color, character size, blink status, intensity, and output format characteristics associated with C&D display information.

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.

Syntax



example:
INITIAL (2#(1, 3#5))

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.

3. A simple <initial list> of the form given in Rule 2 may be 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 g. These items are:

   - a single literal,
   - a single unsubscripted variable name,
   - [<dcw value list>], and
   - blank (i.e., the component(s) of the <identifier> should not be initialized).

5. The presence of an asterisk at the end of the <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. An <expression> must be an unarrayed integer or scalar expression computable at compile time. Type conversion between integer and scalar is allowed where necessary.

Rules for Vector and Matrix Types

1. If the <identifier> has no array specification, then one of the following must hold:

   o the number of values in the <initial list> is exactly one, in which case all components of the vector or matrix are initialized to that value;

   o the number of values in the <initial list> is exactly equal to the number of components to be initialized; or

   o 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:

   o the number of values in the <initial list> is exactly one, in which all the components of all the array elements of the vector or matrix are intitialized to that value;

   o 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;

   o the number of values in the <initial list> is equal to the total number of components in all array elements; or

   o 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. An <expression> must be an unarrayed integer or scalar expression computable at compile time. Type conversion between integer and scalar is allowed where necessary.

Rules for Bit, Boolean, Event, Flag 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:

   o the number of values in the < initial list> is exactly one, in which case all elements of the array are initialized to that value;

   o the number of values in the <initial list> is exactly equal to the number of array elements to be initialized; or

   o 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 or Boolean type is being initialized, <expression> must be an unarrayed <bit exp> computable at compile time.

4. If an <identifier> of character type is being initialized, <expression> must be an arrayed <char exp> computable at compile time.

5. Event types may not be initialized. They are implicitly initialized to zero or a "false" condition.

6. Flag types may not be initialized.

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:

   o the number of values in the <initial list> is equal to the total number of data elements in the whole structure; or

   o 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:

   o 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;

   o the number of values in the <initial list> is equal to the total number of data elements in all copies of the structure;

- o 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 the copies of the structure, and partial initialization is indicated.

3. 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).

Rules for DCW Types

1. If the <identifier> has no array specification and is to assume a default <dcw value list>, no <initial list> is needed. The <dcw value list> defaults to: GREEN, 5MM, BLINK OFF, 4, and EBCDIC INTEGER.

2. If the <identifier> has no array specification, the <initial list> must contain exactly one <dcw value list>.

3. If the <identifier> has an array specification, then one of the following must hold:

   - o the number of <dcw value list> s in the <initial list> is exactly one, in which case all elements of the array are initialized to that value;

   - o the number of <dcw value list> s is exactly equal to the number of array elements to be initialized; or

   - o the< initial list> ends with an asterisk, in which case the number of <dcw value list> s must be less than the number of array elements to be initialized, and partial initialization is indicated.

5. DATA REFERENCING CONSIDERATIONS

Central to the HAL/SM 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 <g var name> primitive, where g stands for any of the types arithmetic, bit, character, dcw, 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 < g var name > , where g stands for any of the types arithmetic, bit, character, or dcw as specified in its < attributes > in the template.

- If a 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 < g var name > , where g stands for any of the types arithmetic, bit, character, or dcw, as specified in its < attributes > in the template.

  Example:

```
STRUCTURE A:
     1 B,
       2 C,
         3 E VECTOR(3),
         3 F SCALAR,              structure template
       2 G,
         3 H BIT(1),
         3 I INTEGER,
     1 J BIT(16);

DECLARE A A-STRUCTURE,     - "unqualified"
        Z A-STRUCTURE;     - "qualified"
```

A. references to parts of structure A -

G I J

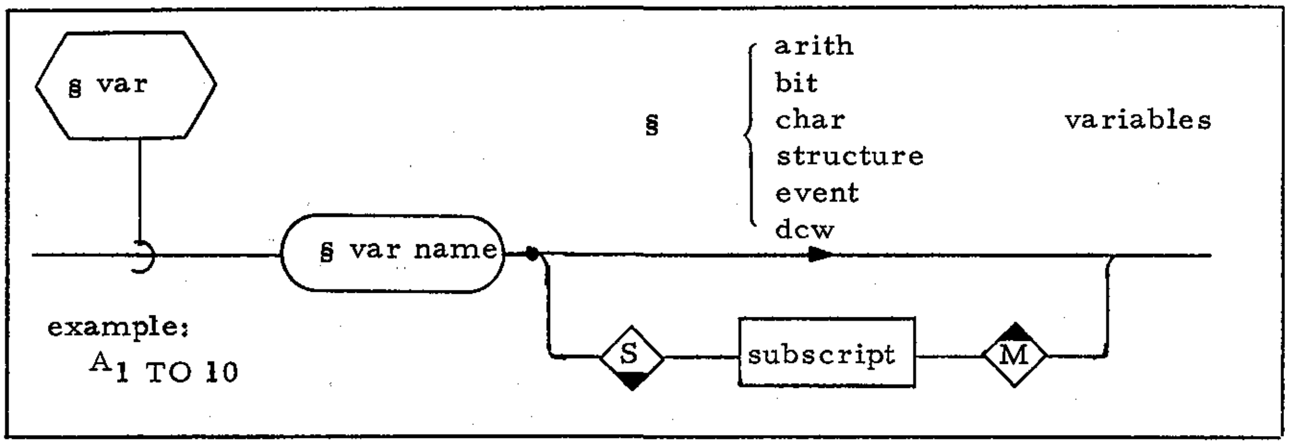
B. references to corresponding parts of structure Z -

Z. B. G Z. B. G. I Z. J

5.3 Subscripting

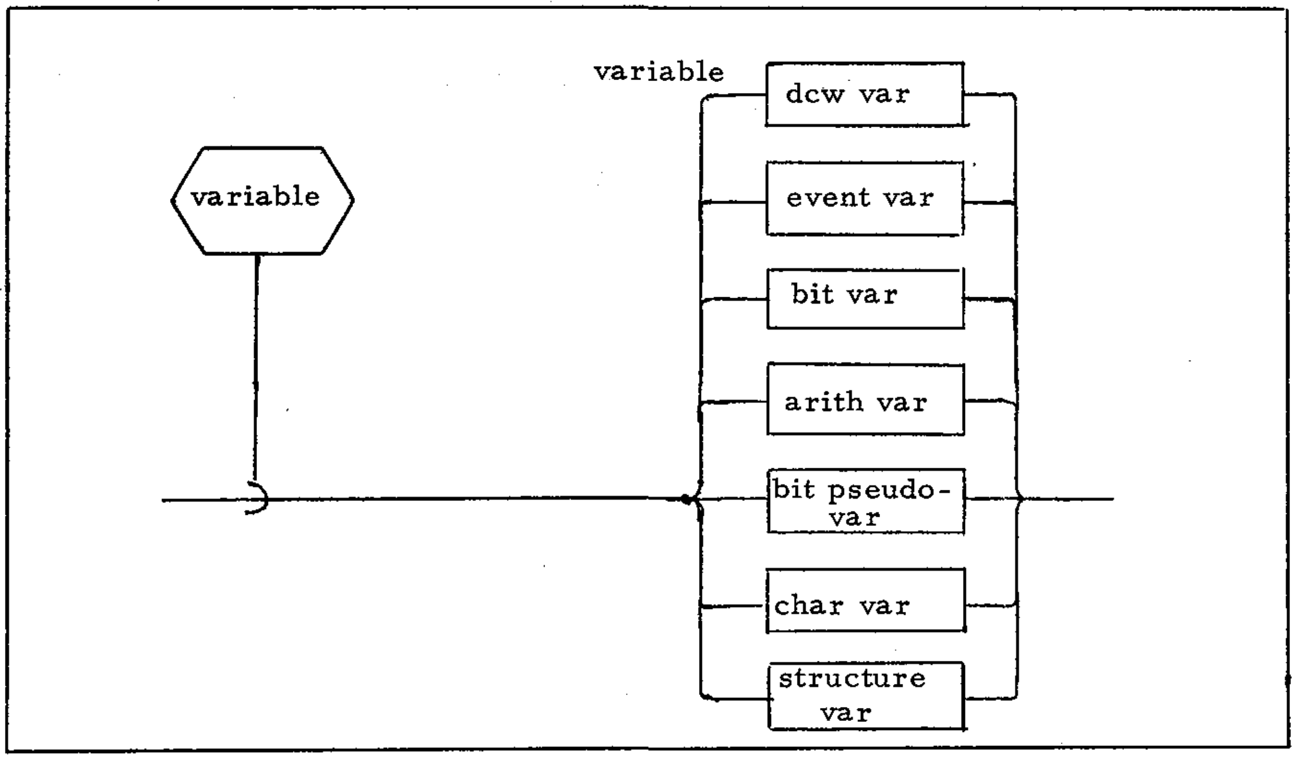
For the remainder of this section, a data name with known <attributes> is denoted syntactically by < ß var name> , where ß stands for any of the types arithmetic, bit, character, event, dcw or structure. It is convenient to introduce the syntactical term <ß var> to denote any subscripted or unsubscripted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

Syntax



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

Syntax



Semantic Rules

1. A <bit pseudo-var> is a reference to the SUBBIT pseudo-variable. An explanation of its inclusion as a <variable> is given in Section 6.6.4.

5.3.1 Classes of Subscripting

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

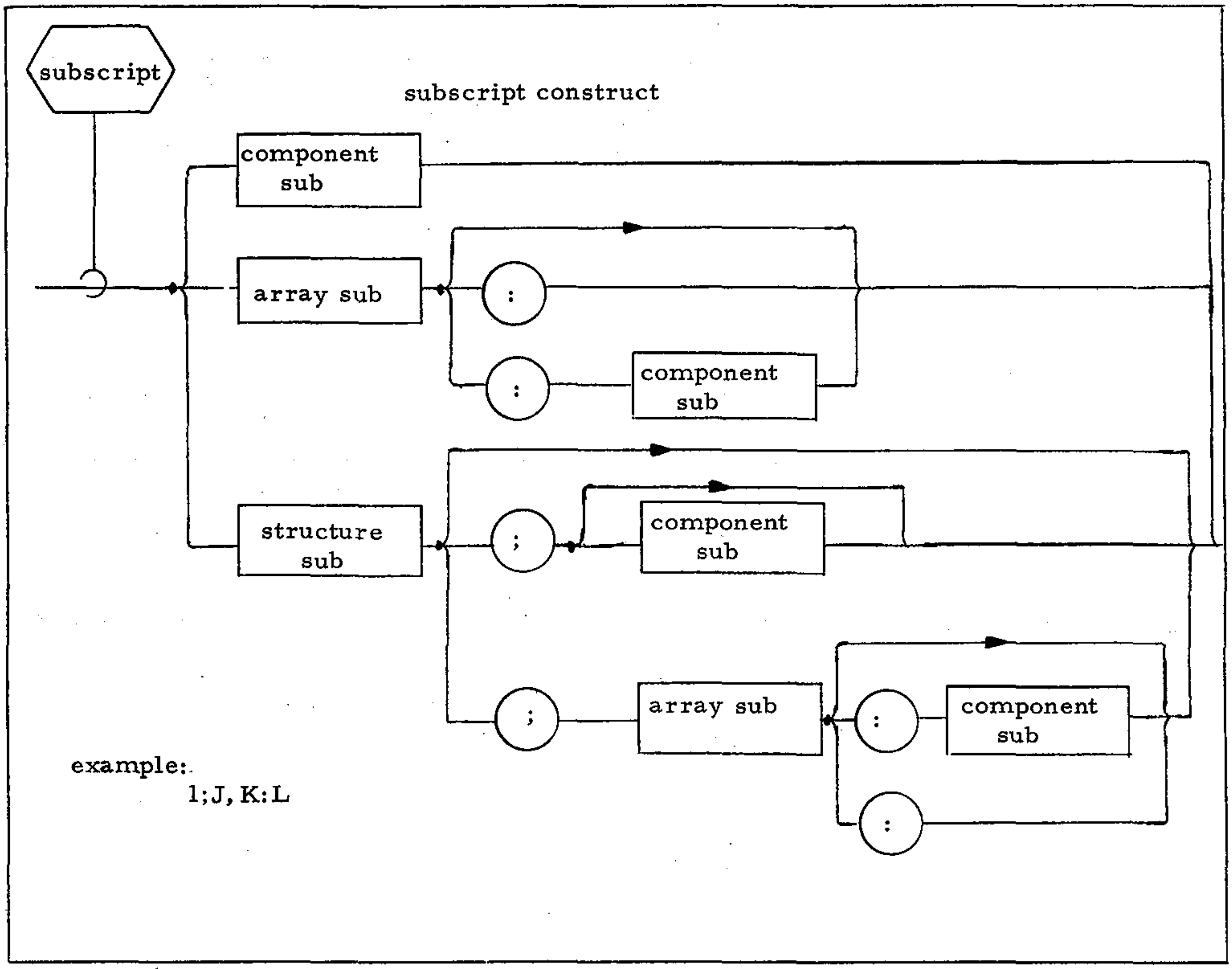
- o Structure subscripting can be applied to arithmetic, bit, character and dcw variables which are terminals of a structure which has multiple copies. It can also be applied to the major and minor structure variable names of such a structure. Structure subscripting is denoted syntactically by <structure sub>.

- o Array subscripting can be applied to any arithmetic, bit, character, dcw, 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>.

- o 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



Semantic Rules

1. The syntax diagram shows 10 different ways of combining the three classes of subscripting. Table 5-1 shows when each of these combinations is legal for simple variables and structure terminals.

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.

LEGAL COMBINATIONS FOR SIMPLE VARIABLES AND STRUCTURE TERMINALS

Data Type	Interpretation of < § var name>			
	Unarrayed Simple Variable	Arrayed Simple Variable	Unarrayed Structure Terminal (1)	Arrayed Structure Terminal (1)
INTEGER SCALAR DCW	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
EVENT	none	A		

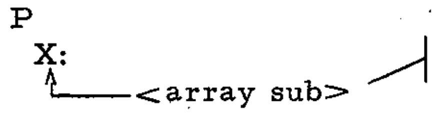
(1) It is assumed that the structure has multiple copies. If not, corresponding columns for simple variables apply.

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

Table 5-1

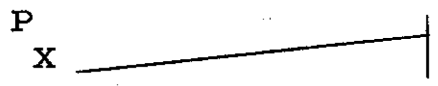
Examples:

A.



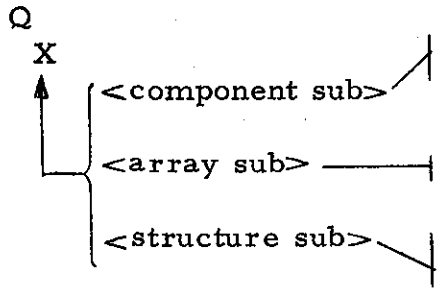
P is any arrayed simple variable

equivalent form -



equivalent only if P is of integer, scalar, or event type

B.

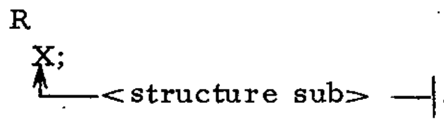


Q is any simple variable

see example A

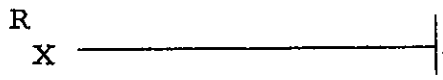
Q is any structure terminal* of integer or scalar type

C.



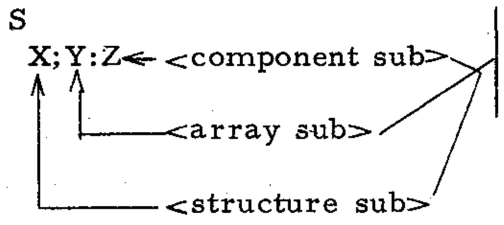
R is any structure terminal*

equivalent forms -



equivalent only if R is of unarrayed integer or scalar type

D.



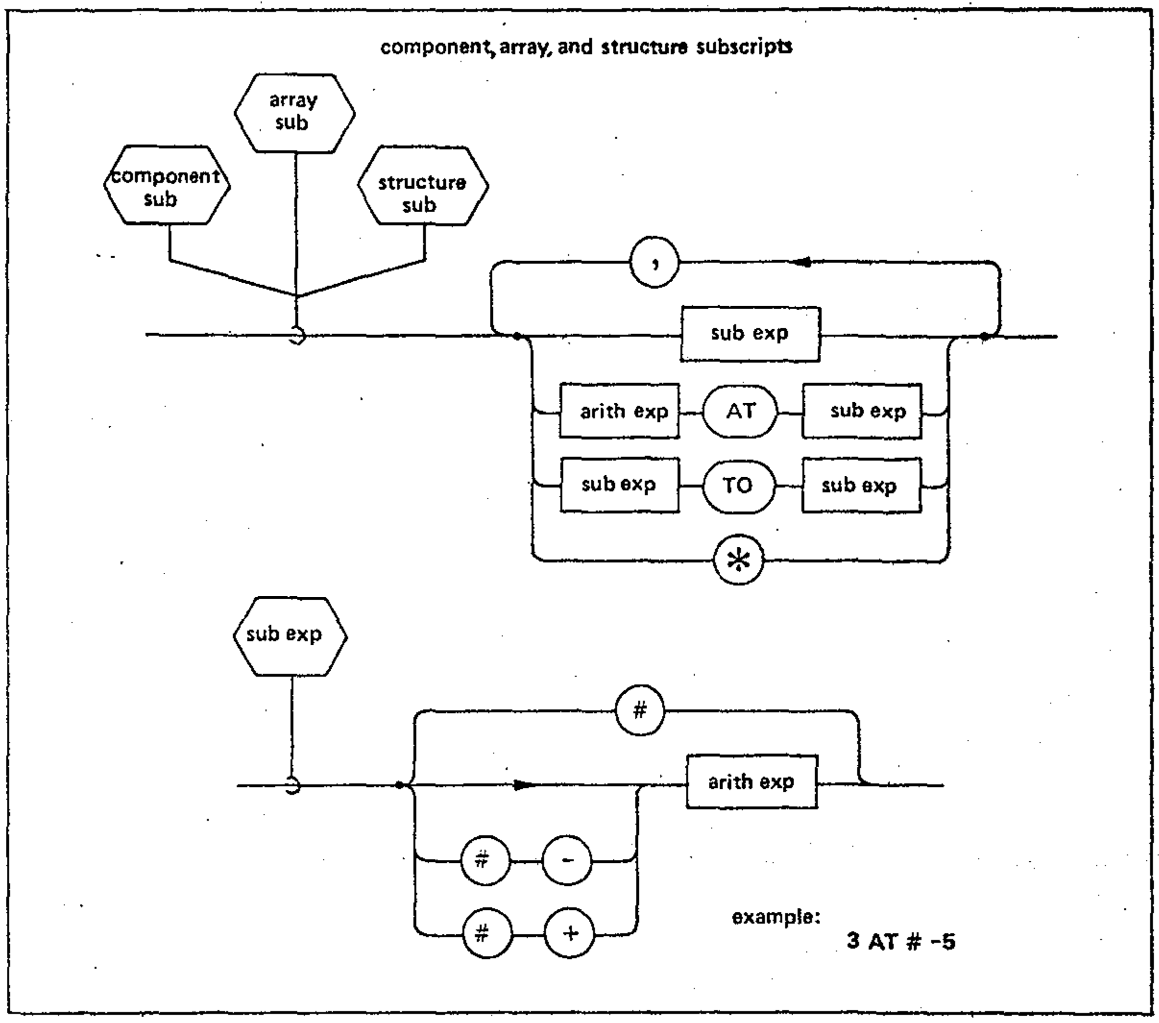
S is an arrayed structure terminal* of vector, matrix, bit, or character type

* of a structure with multiple copies

5.3.2 The General Form of Subscripting

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

Syntax



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 <$_{\S}$ var name> subscripted.

2. There are four forms of subscript expression:

   o the simple index,

   o the AT-partition,

   o the TO-partition, and

   o 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. A <sub exp> may take any of the forms shown. The value of # is taken to be the maximum index-value in the relevant dimension.

8. Any <arith exp> in a subscript expression is an arrayed or unarrayed integer or scalar expression. Values are rounded to the nearest integer. The effect of an <arith exp> being arrayed is discussed in Section 5.4.

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:

   o In the form <arith exp> AT <sub exp> , the value of <arith exp> must be computable at compile time.

   o 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}	20th copy of A
$A_{2 \text{ AT } 10;}$	10th and 11th copies of A (semicolon optional)
C_1	C from 1st copy of A
$D_{4 \text{ TO } 6;}$	D from 4th through 6th copies of A (semicolon enforced)
Note: $D_{*; 4 \text{ TO } 6}$	components 4 through 6 of D from all copies of A

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:

   o In the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time.

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

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.

- o Those of bit, character, and vector types must possess a <component sub> consisting of one subscript expression only.

- o 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 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:

- o In the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time.

- o In the form <sub exp> TO <sub exp>, the values 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)

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:

- o an array specification appears in its declaration in a structure template, or
- o 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/SM 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.

Example:

```
STRUCTURE Z:
      1 B ARRAY(5);
DECLARE A Z-STRUCTURE(10);
DECLARE C ARRAY(10, 5);
.
.
.                                  arrayness of both operands is 10, 5
C = A. B + C;
```

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, except for references to event variables.

2. If the < § var name> possessing the subscript containing the arrayed < arith exp> is imbedded in an arrayed HAL/SM 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:

$$\left.\begin{array}{l} V_A \text{ - a 3-array} \\ V_B \text{ - a 3x2-array} \end{array}\right\} \text{ of corresponding vector components}$$

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 $M_{I,*} \equiv \begin{pmatrix} M_{2,*} \\ M_{1,*} \\ M_{1,*} \end{pmatrix} \equiv \begin{pmatrix} [\ .75 \quad 1.25] \\ [1.75 \quad 0.25] \\ [1.75 \quad 0.25] \end{pmatrix}$

- 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 $MA \equiv \begin{pmatrix} \begin{bmatrix} 1.0 & 0.0 \\ 3.0 & 2.0 \end{bmatrix} \\ \\ \begin{bmatrix} 4.0 & 7.0 \\ 6.0 & 5.0 \end{bmatrix} \\ \\ \begin{bmatrix} 8.0 & 3.0 \\ 4.0 & 9.0 \end{bmatrix} \end{pmatrix}$ $\quad \begin{matrix} \textcircled{1} \; I_1 = 2 \\ \textcircled{2} \; I_2 = 1 \\ \textcircled{3} \; I_3 = 1 \end{matrix}$

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

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 $MB \equiv \begin{pmatrix} \begin{bmatrix} 0.5 & 0.5 \\ 0.1 & 0.3 \end{bmatrix} \\ \\ \begin{bmatrix} 0.2 & 0.7 \\ 0.4 & 0.8 \end{bmatrix} \end{pmatrix}$ $\quad \textcircled{1} \; I_1 = 2$

then $MB_{*:I_{1\ TO\ 2},*} \equiv \begin{pmatrix} MB_{1:2,*} \\ MB_{2:1,*} \end{pmatrix} \equiv \begin{pmatrix} [0.1 & 0.3] \\ [0.2 & 0.7] \end{pmatrix}$ $\textcircled{2} \; I_2 = 1$

5.5 The Natural Sequence of Data Elements

There are several kinds of operations in the HAL/SM 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. Two major occurrences of these processes are in I/O (see Section 10), and in conversion functions (see Section 6.6).

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 replacing "unraveled" by "reraveled," the natural sequence for reraveling is obtained.

Rules for Major and Minor Structure

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

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

Rules for Other Operands

1. An operand of any type (integer, scalar, vector, matrix, bit, character, dcw or event) may possess arrayness as described in Section 5.4. Each

C.2

dimension of arrayness, starting from the leftmost is unraveled in turn, in order of increasing index.

2. Integer, scalar, bit, character, dcw 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);

o order of unraveling of V is $V_{i,*:*}$, $i = 1, 2$

o order of unraveling of each $V_{i,*:*}$ is $V_{i,j:*}$, $j = 1, 2$

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

(standard HAL/SM subscript notation used)

(BLANK)

6. DATA MANIPULATION AND EXPRESSIONS

An expression is an algorithm used for computing a value. In HAL/SM, 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/SM, expressions are divided into three major classes according to their usage.

- o Regular expressions appear in a very large number of contexts through 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 expressions. They are collectively denoted by <expression>.

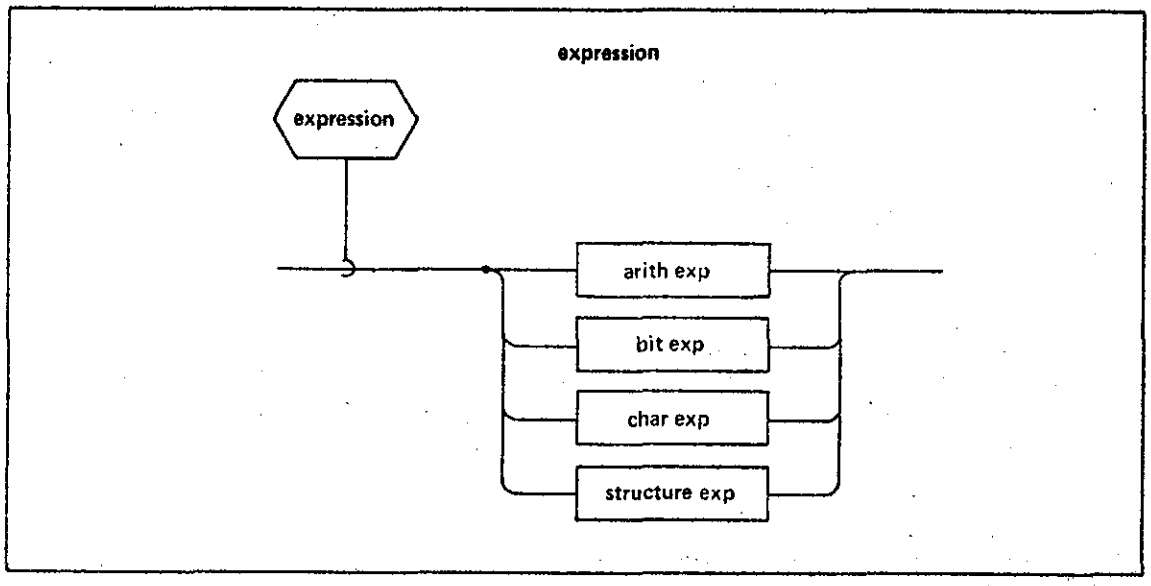
- o Conditional expressions are used to express combinations of relationships between quantities, and are found in IF statements, and in WHILE and UNTIL phrases. They are denoted by < condition> .

- o 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

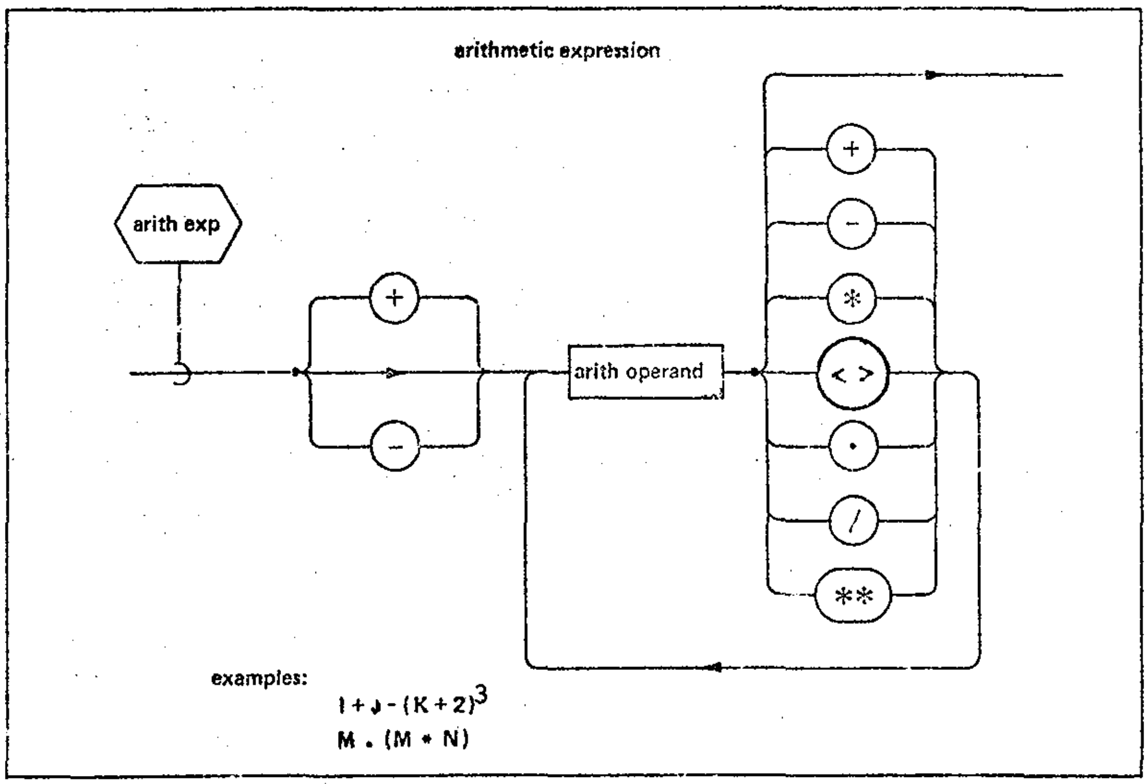


Descriptions of <arith exp> , < bit exp> , < char exp> , and < structure exp > are given in the following subsections.

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



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> are legal in every infix operation. Table 6-1 summarizes all possible forms, by indicating the result of each legal operation.

LEGAL OPERATIONS BY DATA TYPE

operands		infix operator						
left	right	+	-	< >	*	·	/	**
vector	vector	vector	vector	matrix [1]	vector [2]	scalar [3]	▨	▨
vector	matrix	▨	▨	vector	▨	▨	▨	▨
matrix	vector	▨	▨	vector	▨	▨	▨	▨
vector	integer scalar	▨	▨	vector [4]	▨	▨	vector [5]	▨
integer scalar	vector	▨	▨	vector [4]	▨	▨	▨	▨
matrix	matrix	matrix	matrix	matrix	▨	▨	▨	▨
matrix	integer 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 †
integer	scalar	scalar	scalar	scalar	▨	▨	scalar	scalar [8]
integer	integer	integer	integer	integer	▨	▨	scalar [7]	[9]

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 in which case the result is integer.

Table 6-1

5. Except as noted in Rule 4, if only one operand in an operation is of integer type, it is converted to scalar type.

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 same as the precision of the operands. This is true in all cases except where one operand only is of integer type. In this case the precision of the result is the same as the precision of the non-integer operand.

Precedence Rules

1. Table 6-2 summarizes the precedence rules for arithmetic operators.

2. If two operations with the same precedence follow each other, then the following rules apply:

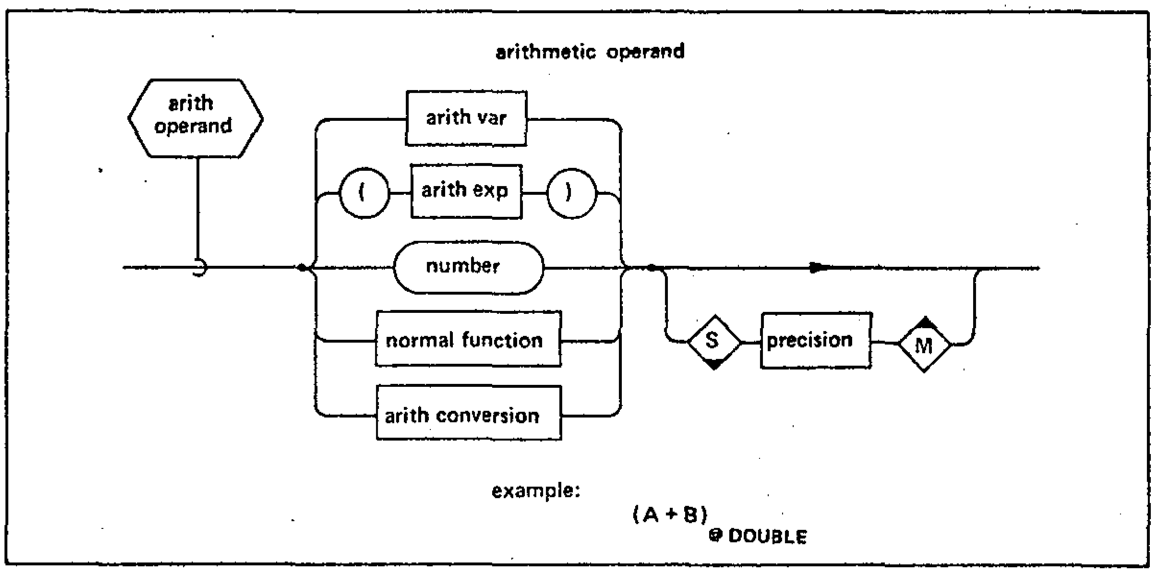
   o operators **, / are evaluated right-to-left;

   o 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



PRECEDENCE RULES FOR ARITHMETIC OPERATORS

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

Table 6-2

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 6.6.1), or a literal <number> .

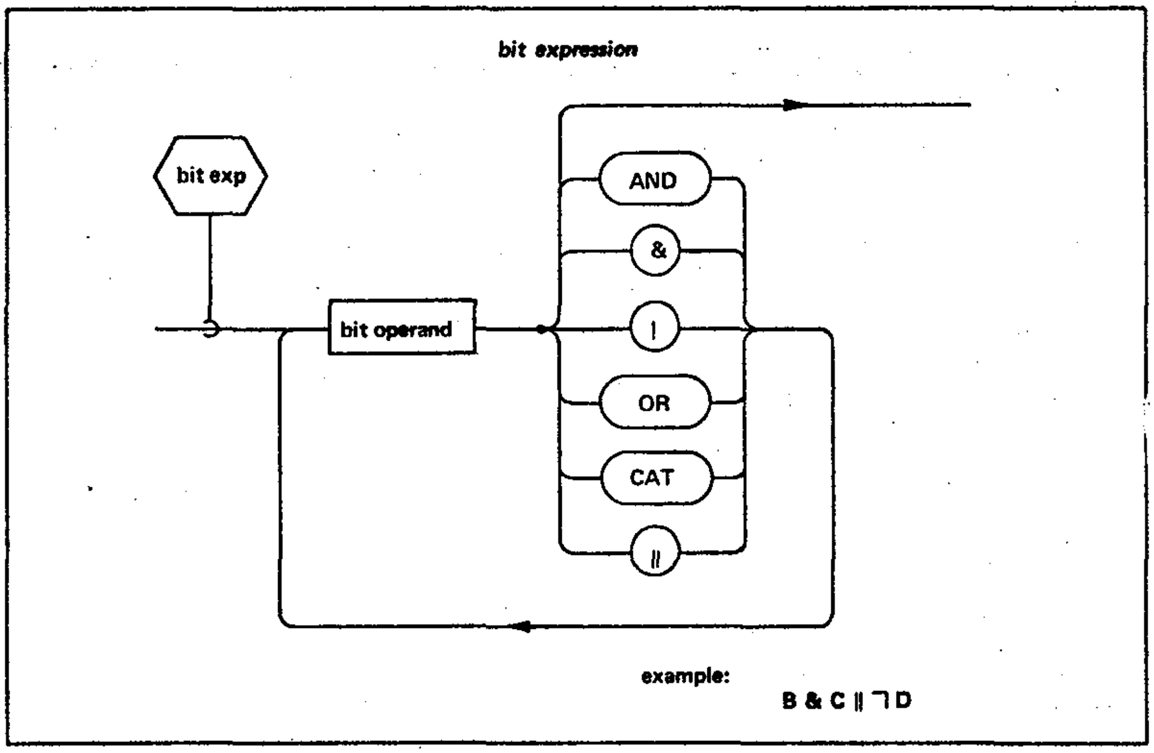
2. The precision of an <arith operand> may be converted by subscripting it with a <precision> specifier (see Section 6.7). If the operand is an< arith var> this is true only if it has no <subscript> . Since a subscripted <arith var> is an example of an <arith exp> , the <precision> specifier may be applied by first enclosing the <arith var> in parentheses.

3. Only integer and scalar <arith operand> s may have the form <number> .

6.1.2 Bit Expressions

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

Syntax



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

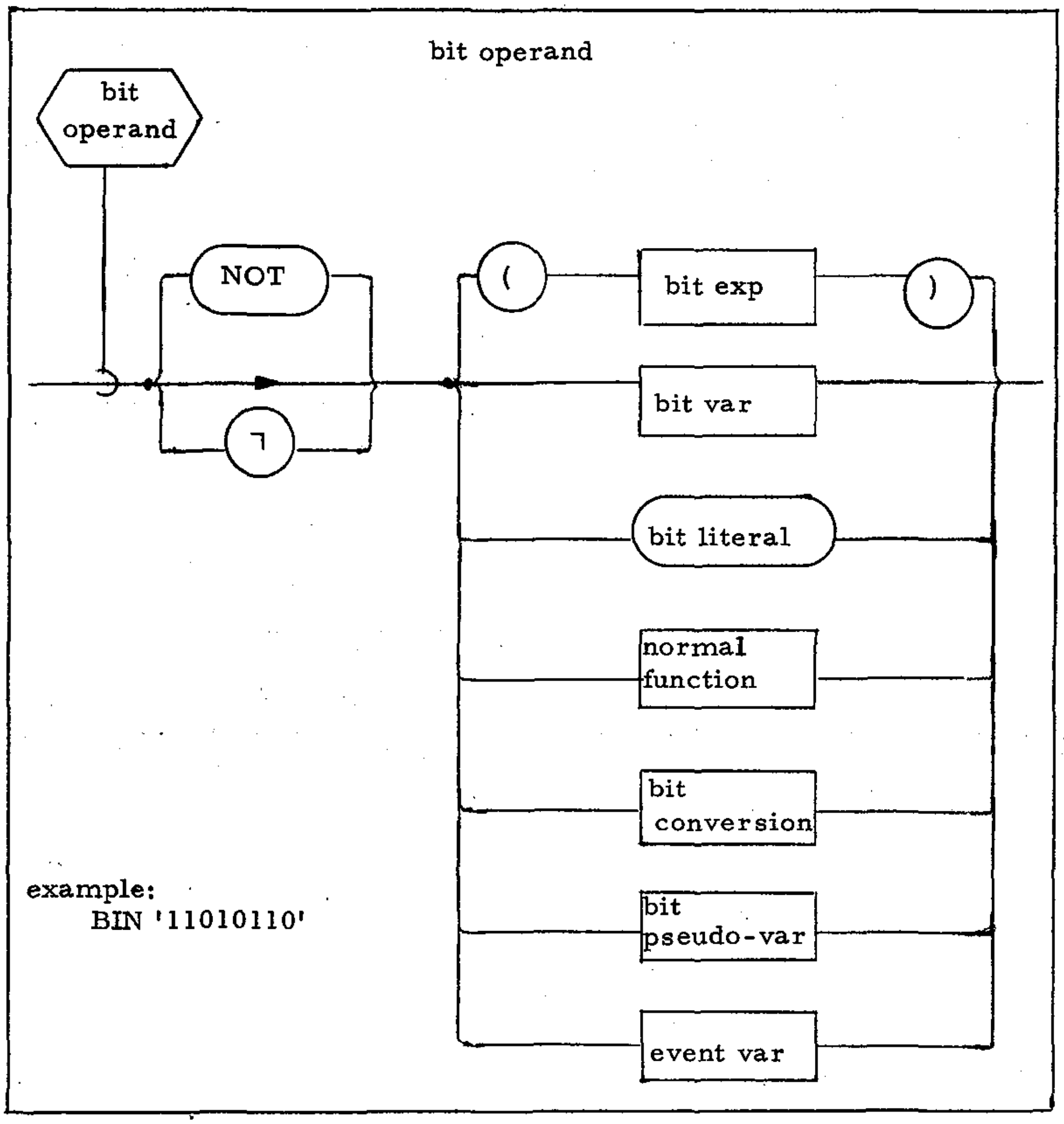
If two operations with the same precedence follow each other, they are evaluated from left to right.

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> 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



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 Section 6.6.3 and 6.6.4).

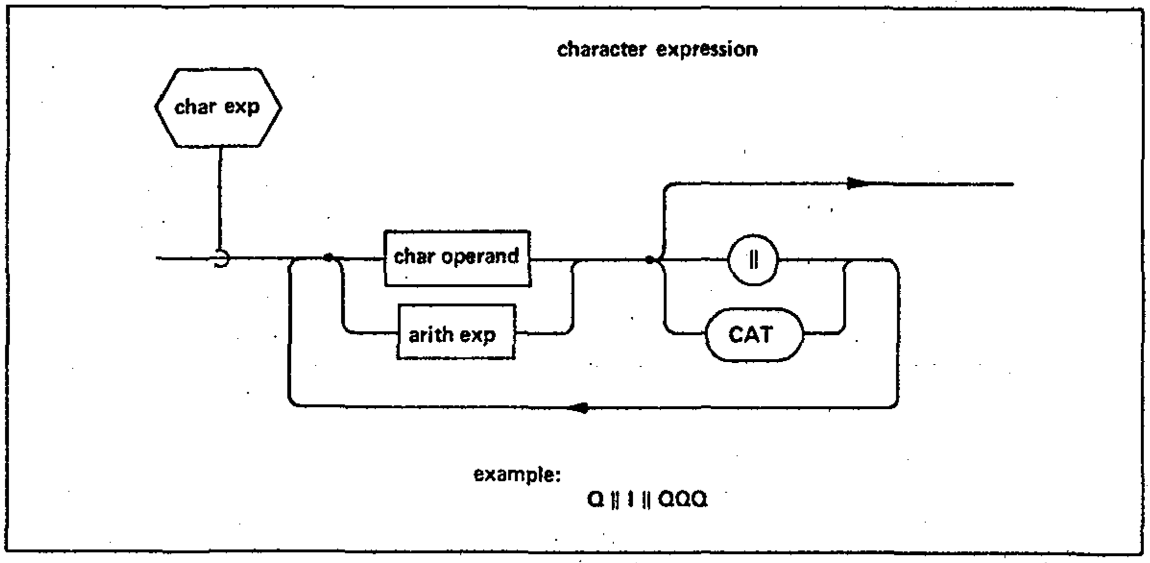
2. In addition a <bit operand> may be an <event var> . Events are treated as Boolean (1-bit) <bit operand> s.

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



Semantic Rules

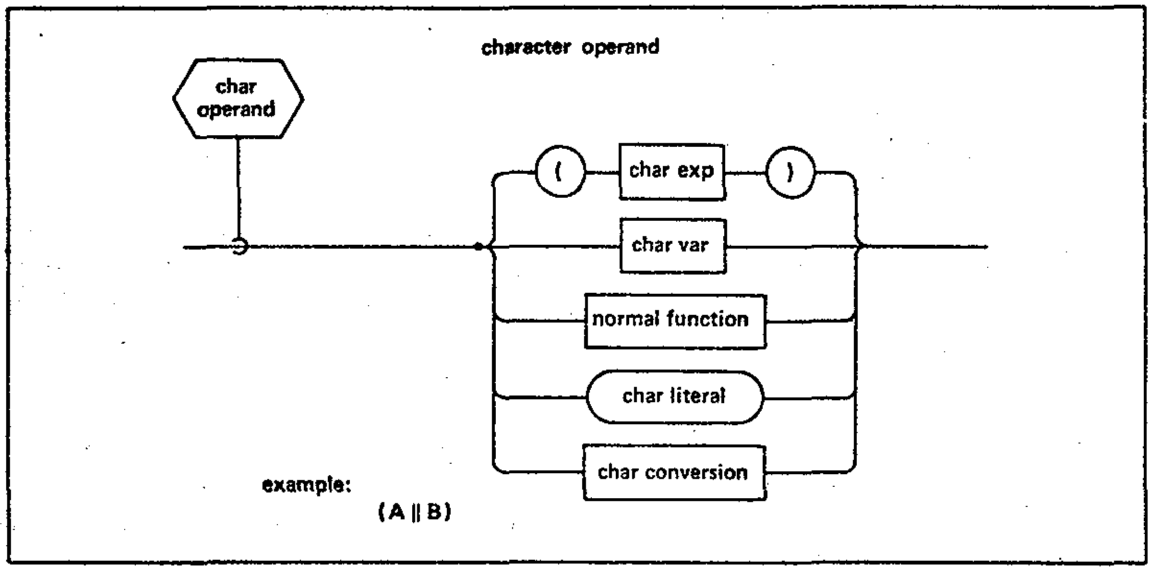
1. A <char exp> is a sequence of operands separated by the catenation operator CAT (||). Each operand may be a <char operand> or an integer or scalar <arith exp> .

2. The sequence of catenations is evaluated from left to right.

3. Integer and scalar <arith exp> s are converted to character strings.

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

Syntax



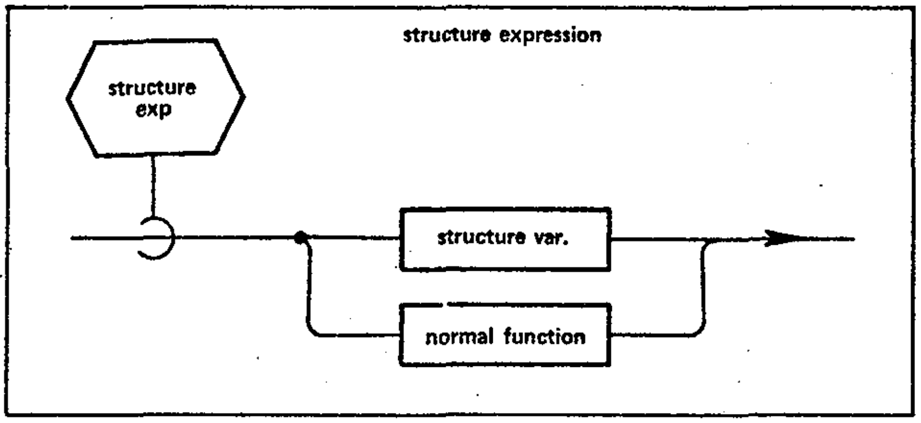
Semantic Rules

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 6.6.3).

6.1.4 Structure Expressions

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

Syntax



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

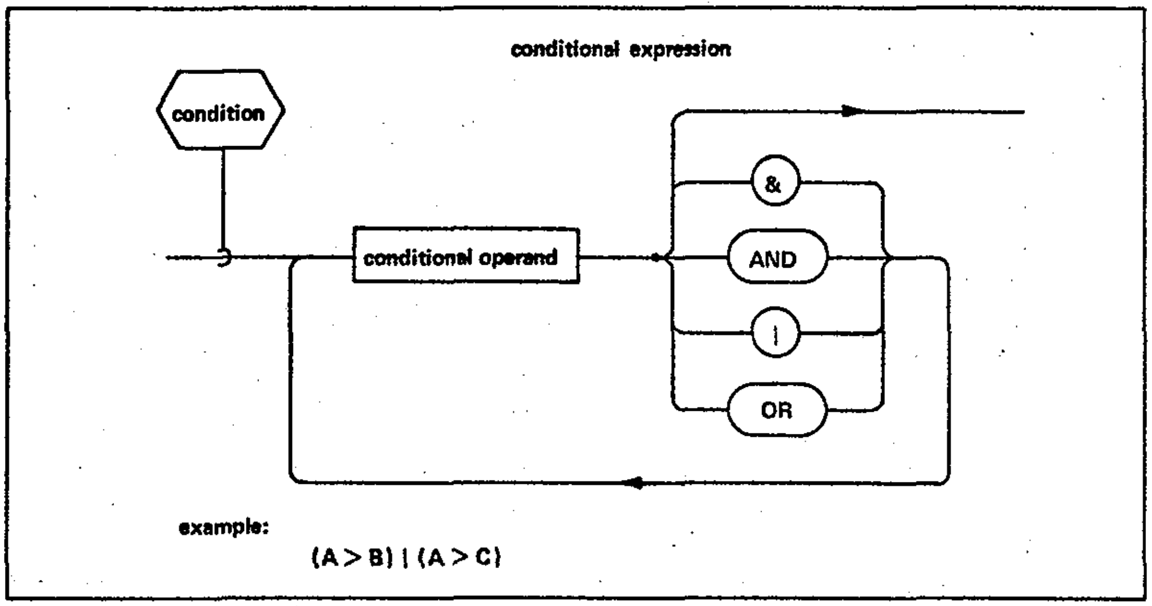
1. If one of the two operands of an infix operation are arrayed, then evaluation of the operation using the unarrayed operand and each element of the arrayed operand is implied. The resulting array has the same dimensions as the arrayed operand.

2. If both of the operands of an infix operation are arrayed, then both operands must have the same array 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/SM representation of a relation between quantities is a <comparison>. The <comparison> s are combined with logical operators to form conditional expressions, or <condition> s.

Syntax



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:

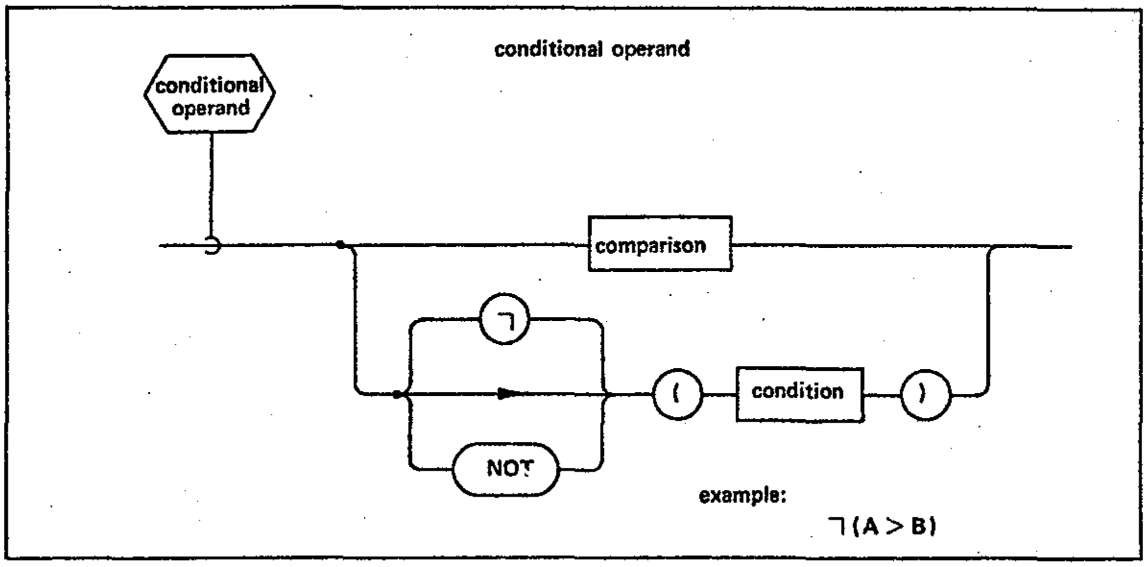
Operator	Precedence
	FIRST
AND, &	1
OR, |	2
	LAST

If two operations with 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



Semantic Rules

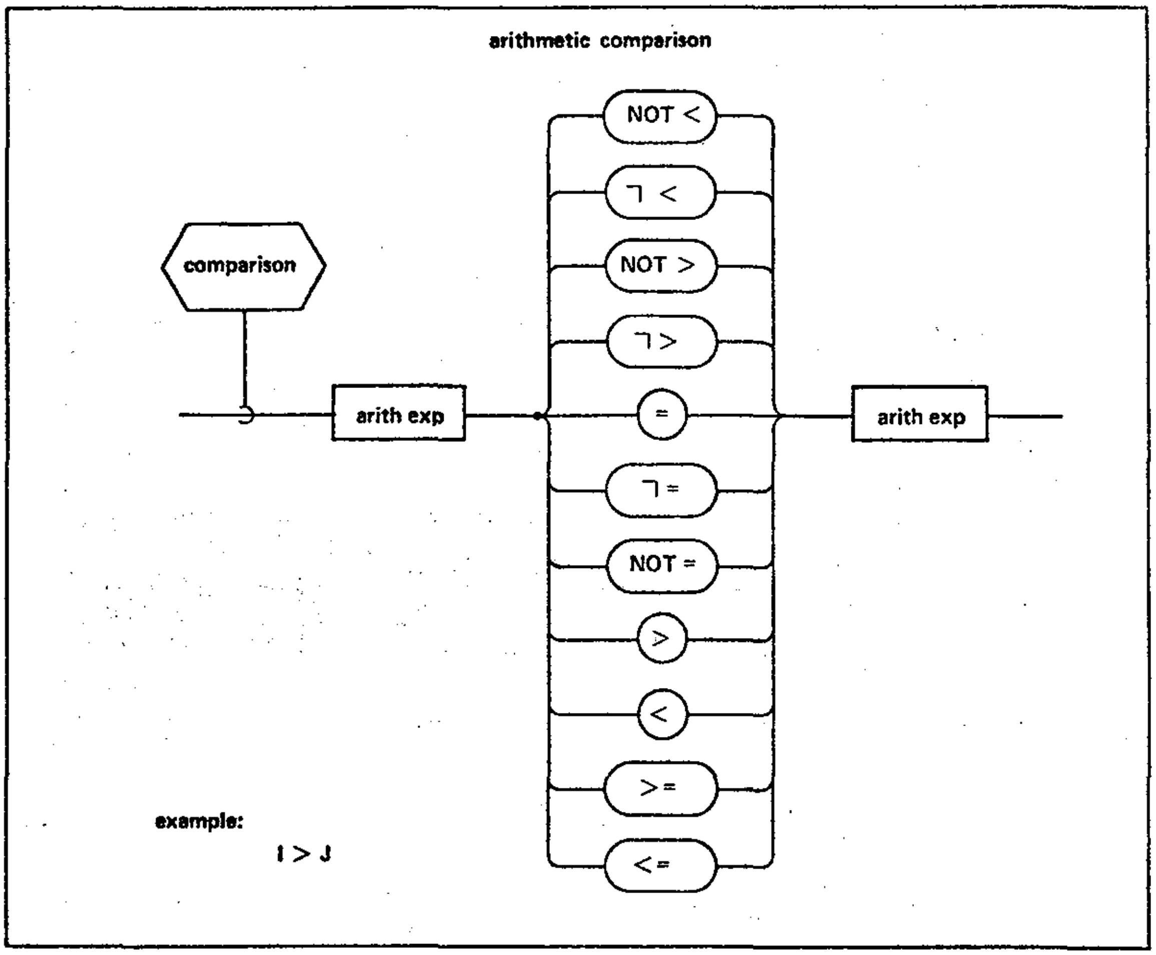
1. A <conditional operand> is either a <comparison> or a parenthesized <condition>. The latter form may be preceded by the logical NOT (¬) operator.

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



Semantic Rules

1. The types of <arith exp> operand must in general match, with the following exception: in a comparison with mixed integer and scalar operands, the integer operand is converted to scalar.

2. If the precisions 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 Table 6-3 indicate all legal forms.

4. If the operands are of vector or matrix type, the <comparison> is carried out on an element-by-element basis.

LEGAL COMPARISONS BY DATA TYPE

operands	operator					
	=	¬= NOT=	>	<	¬> NOT> <=	¬< NOT< >=
vector	✓	✓	▨	▨	▨	▨
matrix	✓	✓	▨	▨	▨	▨
integer scalar	✓	✓	← no arrays →			

Table 6-3

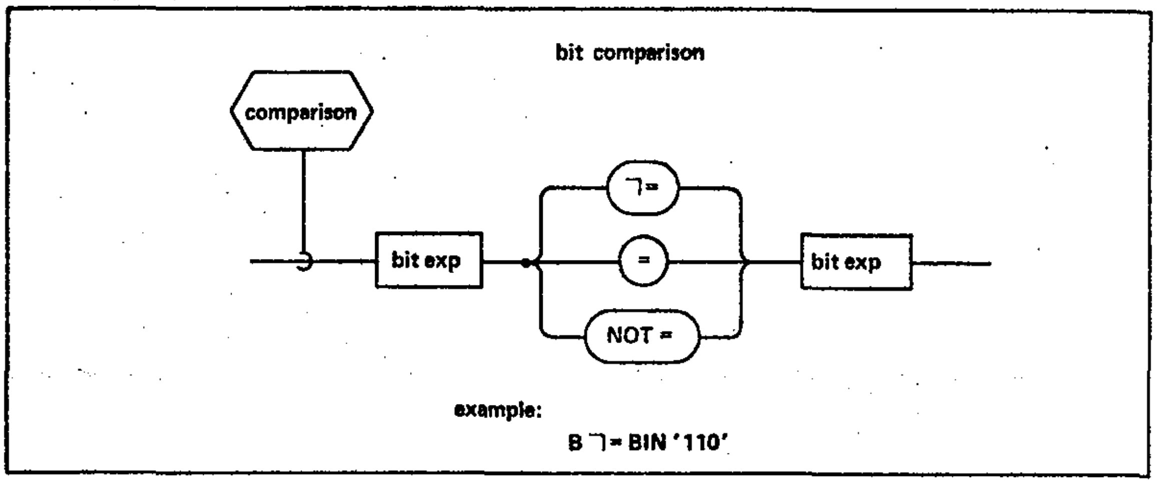
- If the <comparison> operator is =, the result is TRUE only if all the elemental comparisons are TRUE.
- If the <comparison> operator is NOT= (¬=), the result is TRUE if any elemental comparison is TRUE.

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



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.

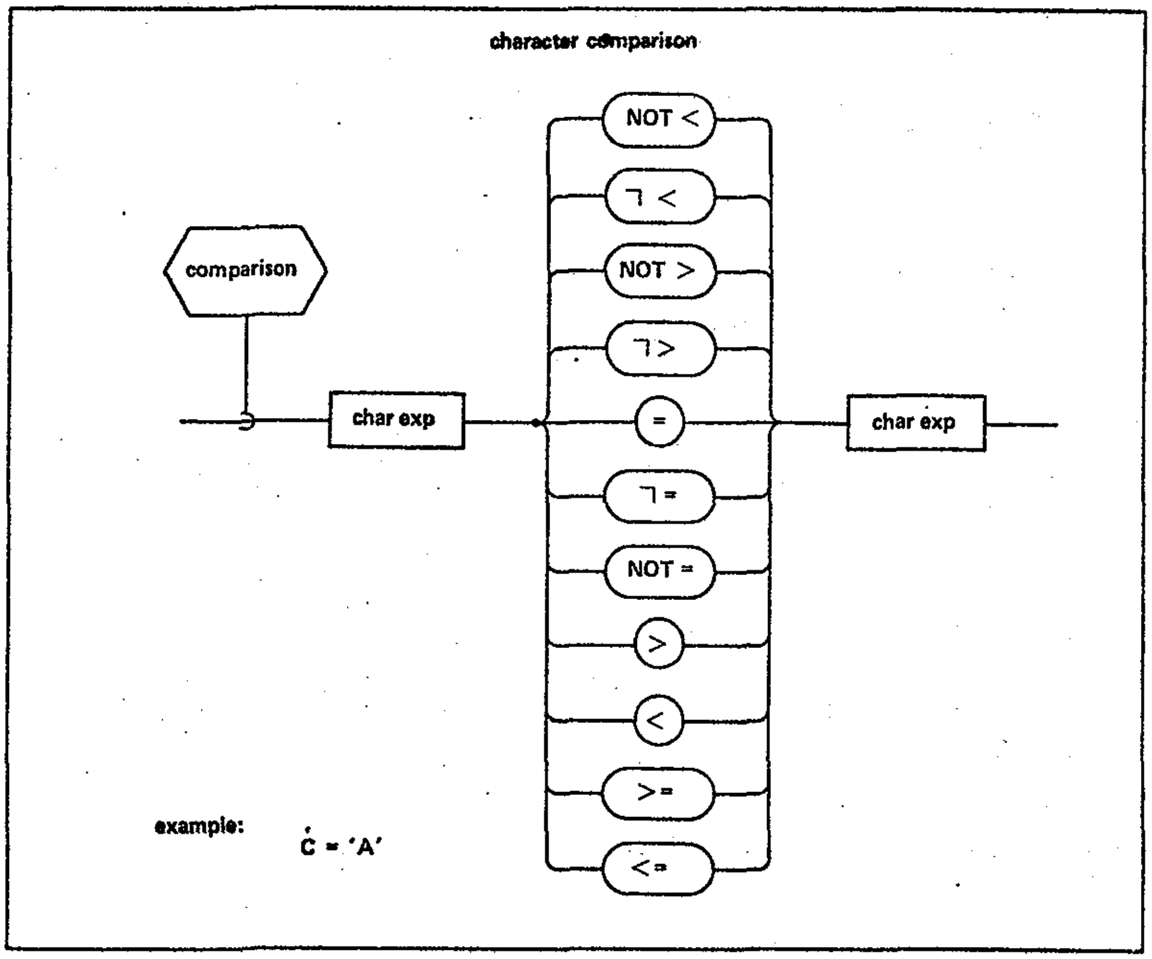
2. If the lengths of the operands differ, the <bit exp> of shorter length is left padded with binary zeros to match the length of the longer before comparison takes place.

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



Semantic Rules

1. If the lengths of the operands differ, the shorter operand is considered less than the longer.

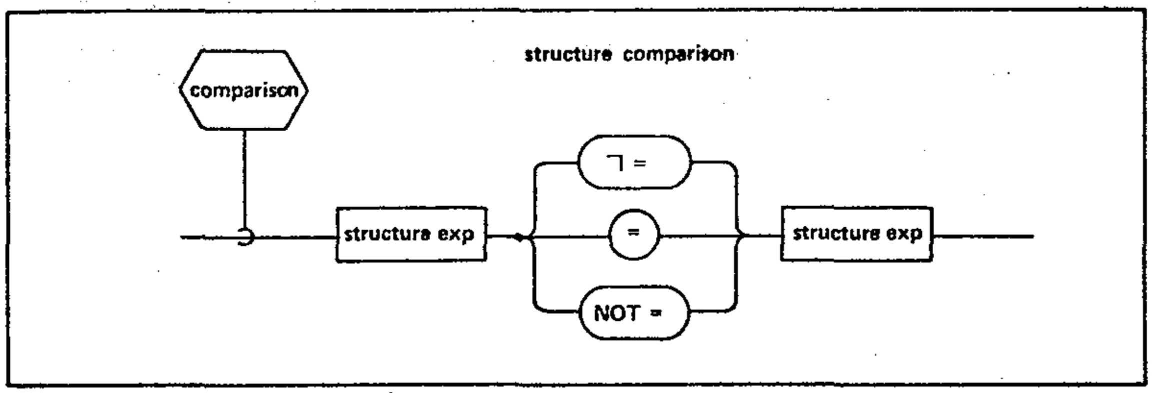
2. If one or both of the <char exp> s are arrayed then the result is an arrayed< comparison> (see Section 6.2.5).

3. The values of the operands will conform to the character codes selected and thus are machine dependent.

6.2.4 Structure Comparisons

A structure comparision is a comparison between two structure expressions.

Syntax



Semantic Rules

1. The tree organizations of both <structure exp> s must be identical in all respects.

2. The number of copies possessed by each <structure exp> must be the same. If the number of copies is greater than one, then the following holds:

   o If the <comparison> operator is =, the result is TRUE only if it is TRUE for all copies.

   o 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.

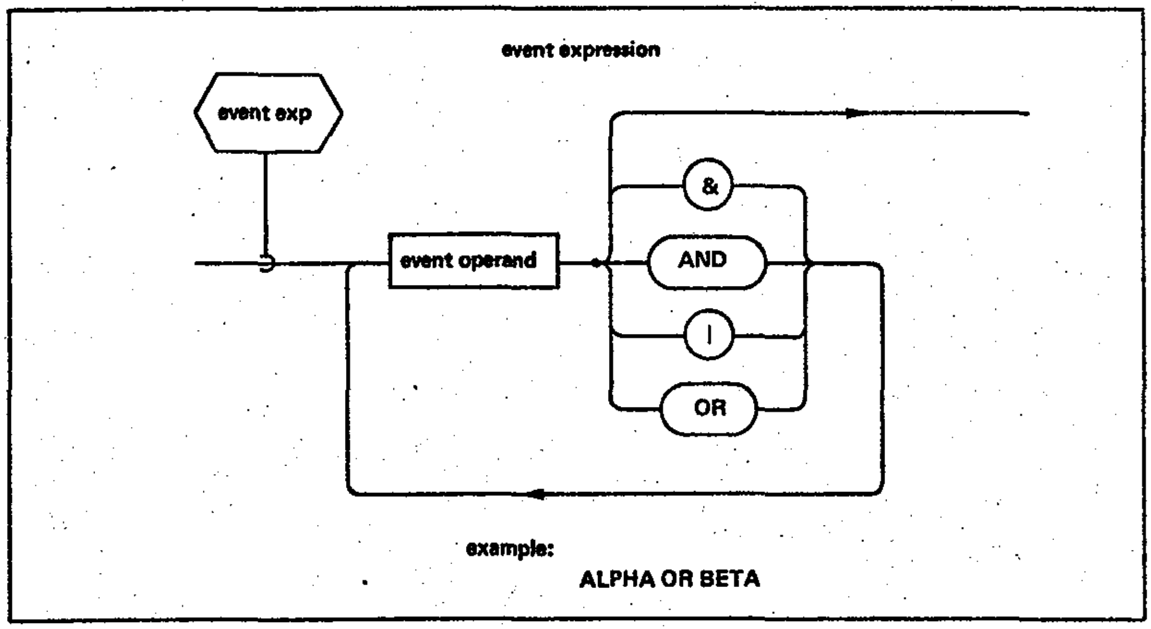
2. If both of the operands are arrayed, then both operands must have the same array dimensions. Evaluation of the operation for each of the corresponding elements of the operands is implied.

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



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:

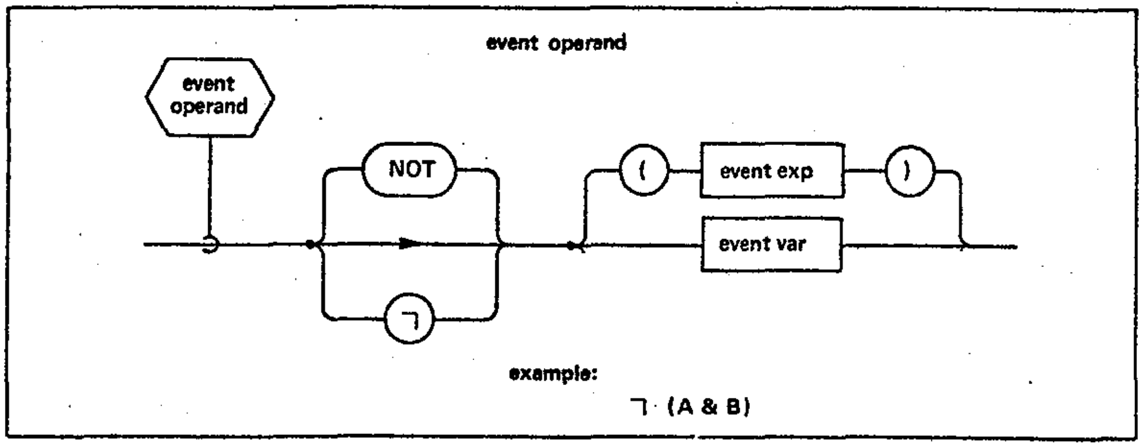
Operator	Precedence
	FIRST
AND, &	1
OR, |	2
	LAST

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.

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

Syntax



Semantic Rules

1. An <event operand> may be an event variable or an <event exp> enclosed in parentheses.

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 (¬).

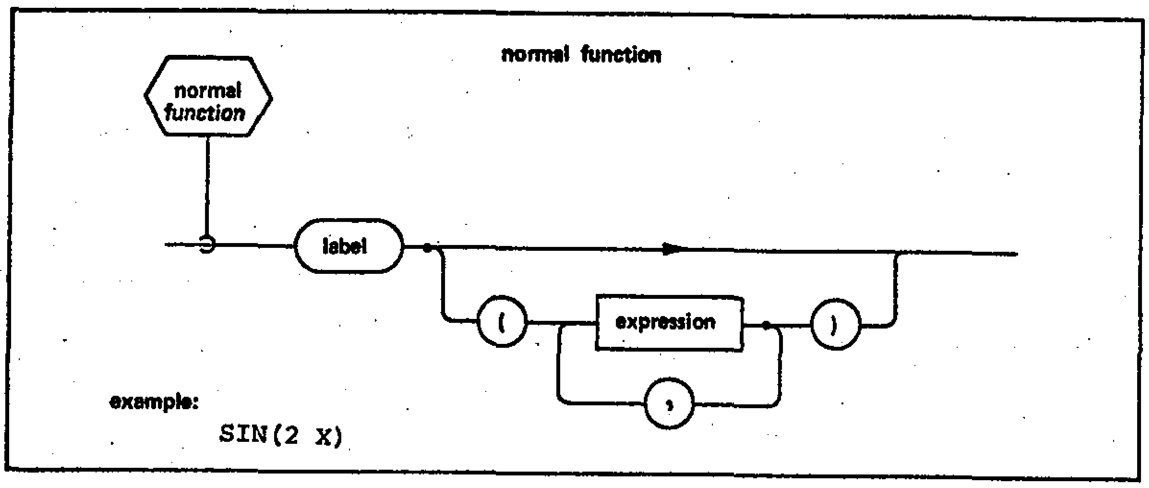
6.4 Normal Functions

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

- o built-in functions defined as part of the HAL/SM language;
- o 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



Semantic Rules

1. A <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.

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. 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).

6. 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" (see Section 7.4).

7. Each input argument of a < normal function> must match the corresponding input parameter of the function definition exactly in type, dimension, structure function, and tree organization, as applicable, except for the following relaxations:

   o precisions need not match, precision conversions are allowed;

   o the lengths of bit arguments need not match;

   o the lengths of character arguments need not match;

   o implicit integer to scalar and scalar to integer conversions are allowed;

   o implicit integer and scalar to character conversions are allowed.

   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.

8. 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).

9. Special considerations relate to arrayed input arguments to the <normal function>. If the corresponding input parameter is arrayed, then the arraynesses 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.

Example:

```
        DECLARE X ARRAY(4) SCALAR;
        .
        .
        .
        [X] = SIN([X]);              SIN evaluated once for each element of X
        .
ADD     FUNCTION (P) SCALAR;
        DECLARE P ARRAY(4) SCALAR;
        .
        .
```

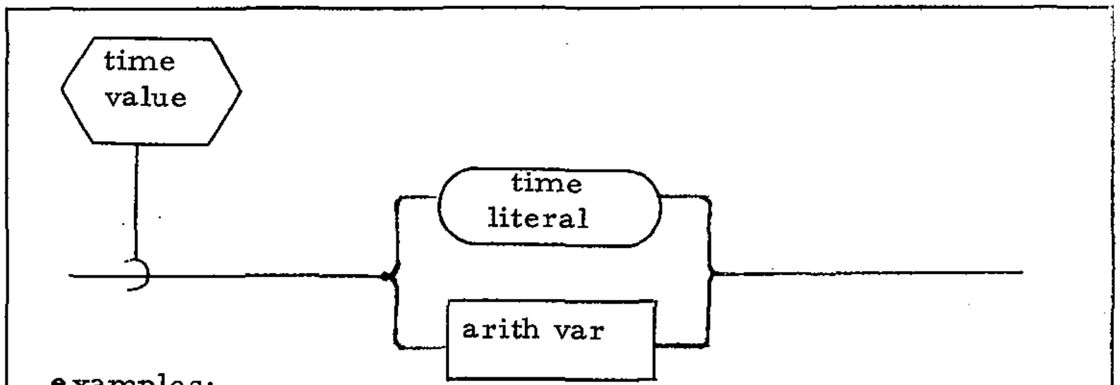
```
RETURN P1+P2+P3;
CLOSE ADD;
.
.
.
.
[X] = [X] + ADD([X]); ——— { ADD evaluated once only: formal
                            parameter P has same arrayness as
                            argument X. [ADD must be defined before
                            its invocation].
```

Note: [ ] enclosing a variable name indicates that it has been declared to be arrayed.

6.5 Time Value

Certain Real Time Control and Input/Output Statements require a time specification. The syntactical term <time value> is used to define that time specification.

Syntax



examples:

START-TIME
50 HRS 15 MINS

Semantic Rules

1. The <time literal> defines a particular time in any combination of days, hours, minutes, seconds, or milliseconds.

2. The <arith var> always defines a particular number of milliseconds.

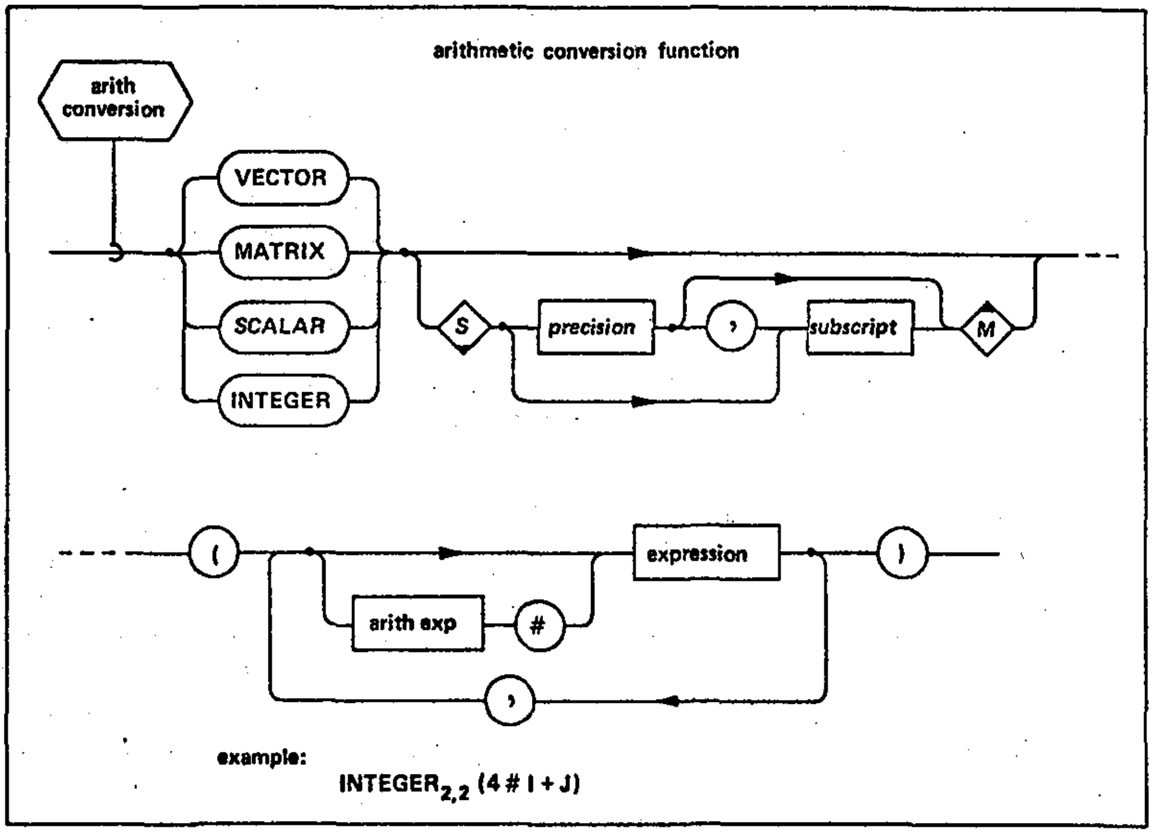
6.6 Explicit Type Conversions

The limited implicit type conversions offered by HAL/SM are described elsewhere in the Specification (see Sections 6.1.1 and 7.3). HAL/SM 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/SM contains conversion functions to integer, scalar, vector, matrix, bit and character types.

6.6.1 Arithmetic Conversion Functions

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

Syntax



General Semantic Rules

1. The keyword INTEGER, SCALAR, VECTOR, or 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 6.7), 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" (See Section 5.5). 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. 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 product must therefore match the total number of data elements implied by the argument(s) of the conversion.

4. INTEGER and SCALAR may have arguments of any type except structure.

5. The precision of the result is SINGLE unless forced by the presence of a <precision> specifier.

Semantic Rules: VECTOR and MATRIX

1. In the absence of <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.

3. In VECTOR the <subscript> must be an <arith exp>. <arith exp> is an unarrayed integer or scalar expression computable at compile time. 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 type 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
```

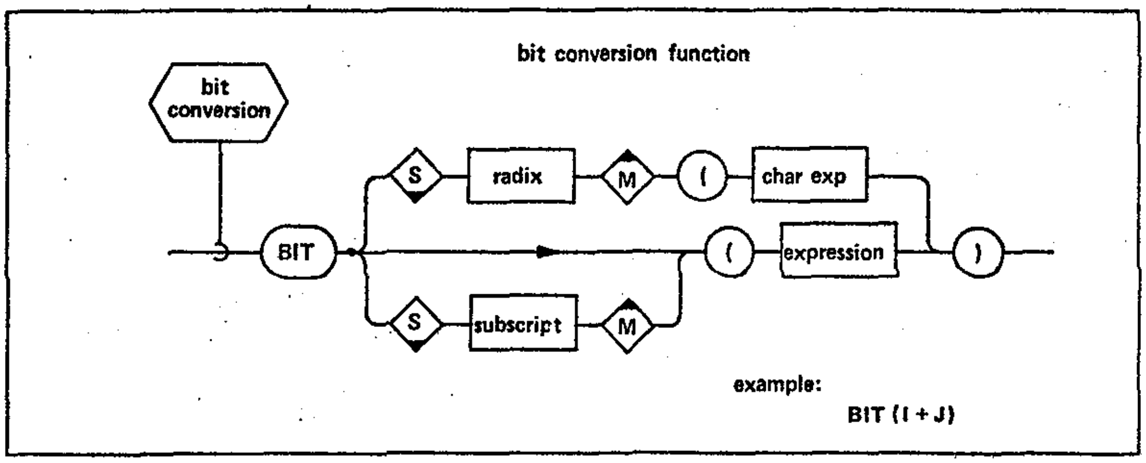
$\text{INTEGER}_{2,6}(2\#[X])$	result is 2, 6 array of integers*
MATRIX(3#V)	result is 3-by-3 matrix, each row being equal to V
$\text{VECTOR}_6([X])$	vector of length 6

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

6.6.2 The Bit Conversion Function

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

Syntax



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 standard conversion rules. The length of the resulting bit string is a fullword, and the significant data is right justified within the word.

---

* For example:

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

1. Argument 2# [X] is "first unraveled," i.e.,
$[\underbrace{1\ 2\ 3\ 4\ 5\ 6}\ \underbrace{1\ 2\ 3\ 4\ 5\ 6}]$

2. Linear string is then "reraveled" into 2x6 array:
$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 6 & 6 \end{bmatrix}$$

2. A <subscript> represents component subscripting upon the results of the conversion. A <subscript> has the same semantic meaning and restrictions in the current context as it does in the subscripting of bit <variable> 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>. A <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;
•
•
•
```

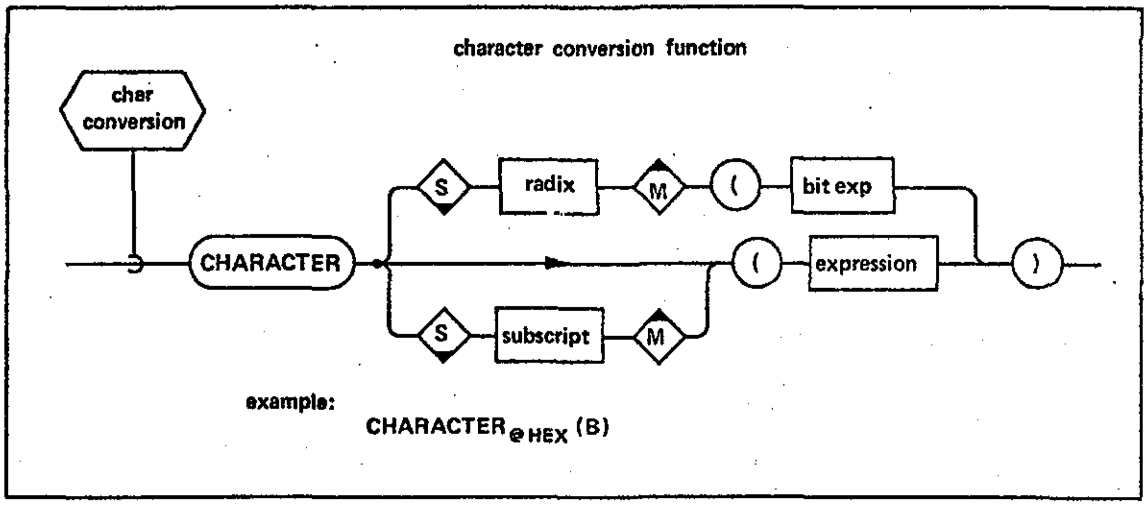
BIT([X])	result is a 2,3 array of bit strings
$BIT_{1\ TO\ 16}([X])$	same as above except that only bits 1 through 16 of each array element are taken
$BIT_{@HEX}$('FACE')	result is bit pattern of hexadecimal digits represented by argument

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

6.6.3 The Character Conversion Function

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

Syntax



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. A <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> . A <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 zeros as required. The result is a character string consisting of the digits of the representation.

Examples:

DECLARE X ARRAY(2, 3) SCALAR;

•

•

•

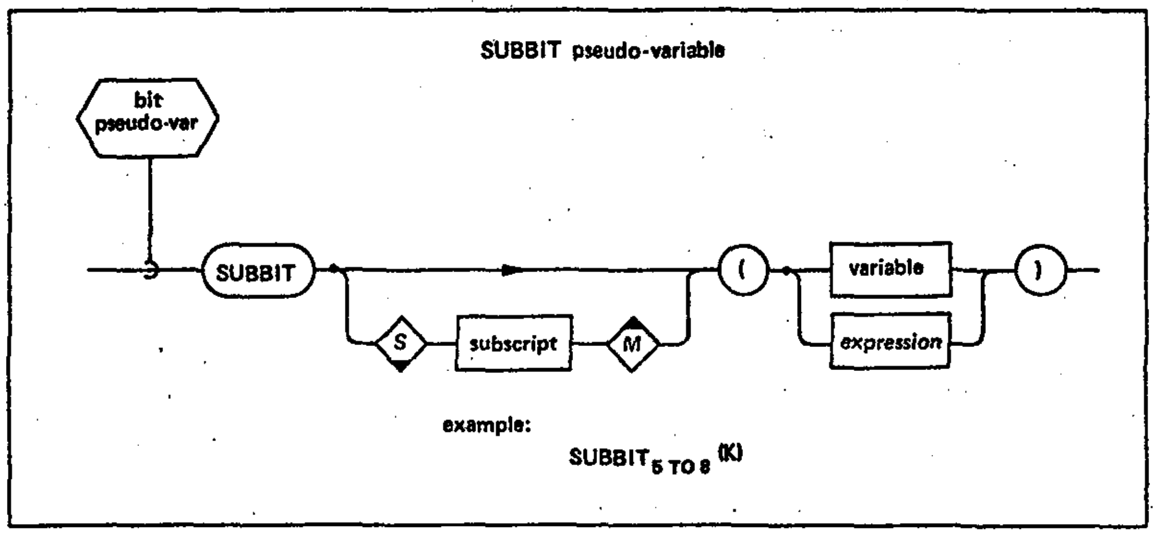
CHARACTER([X])	result is a 2, 3 array of character strings.
CHARACTER_2[(X])	same as above except that only the second character of each array element is taken.
$\text{CHARACTER}_{@\text{DEC}}$(BIN'101101')	result is decimal representation of the bit pattern of the argument.

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

6.6.4 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



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. A <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 zeros 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;
.
.
.
```

$\text{SUBBIT}_{33\ \text{TO}\ 64}(\text{P})$ bits 33 through 64 of the machine representation of P look like a 32-bit bit variable.

bits 1 through 32 are invisible.

6.6.5 Summary of Argument Types

The checkmarks in Table 6-3 indicate the legal argument types for each conversion function.

LEGAL CONVERSIONS

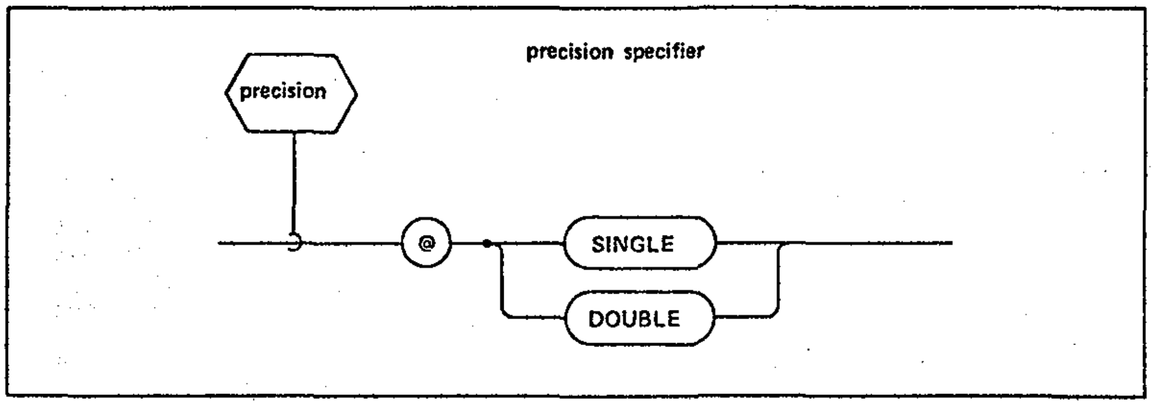
conversion function	argument type					
	integer	scalar	vector	matrix	bit	character
INTEGER	✓	✓	✓	✓	✓	✓
SCALAR	✓	✓	✓	✓	✓	✓
VECTOR	✓	✓	✓	✓	▨	▨
MATRIX	✓	✓	✓	✓	▨	▨
BIT	✓	✓	▨	▨	✓	✓
BIT with <radix>	▨	▨	▨	▨	▨	✓
CHARACTER	✓	✓	▨	▨	✓	✓
CHARACTER with <radix>	▨	▨	▨	▨	✓	▨
SUBBIT	✓	✓	▨	▨	✓	✓

Table 6-4

6.7 Explicit Precision Conversion

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

Syntax



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.

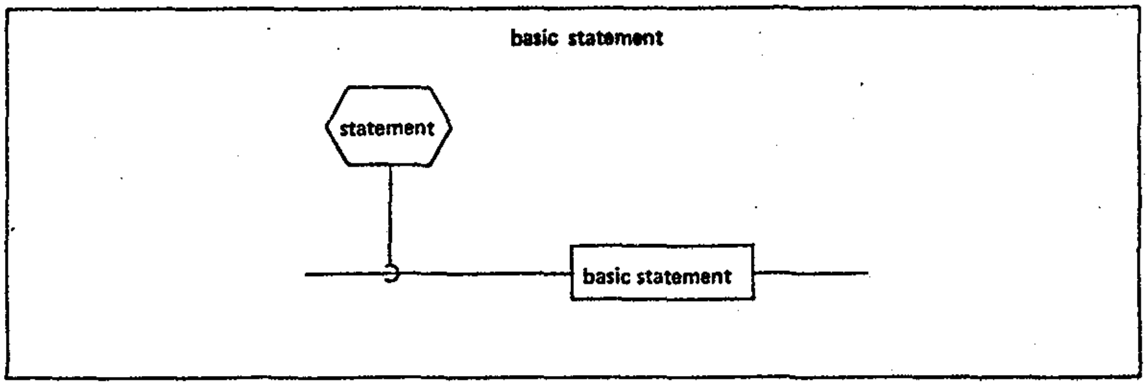
7. EXECUTABLE STATEMENTS

Executable statements are the building blocks of the HAL/SM 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 a HAL/SM 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 (Section 9.1), fall into the category of a <basic statement>.

Syntax

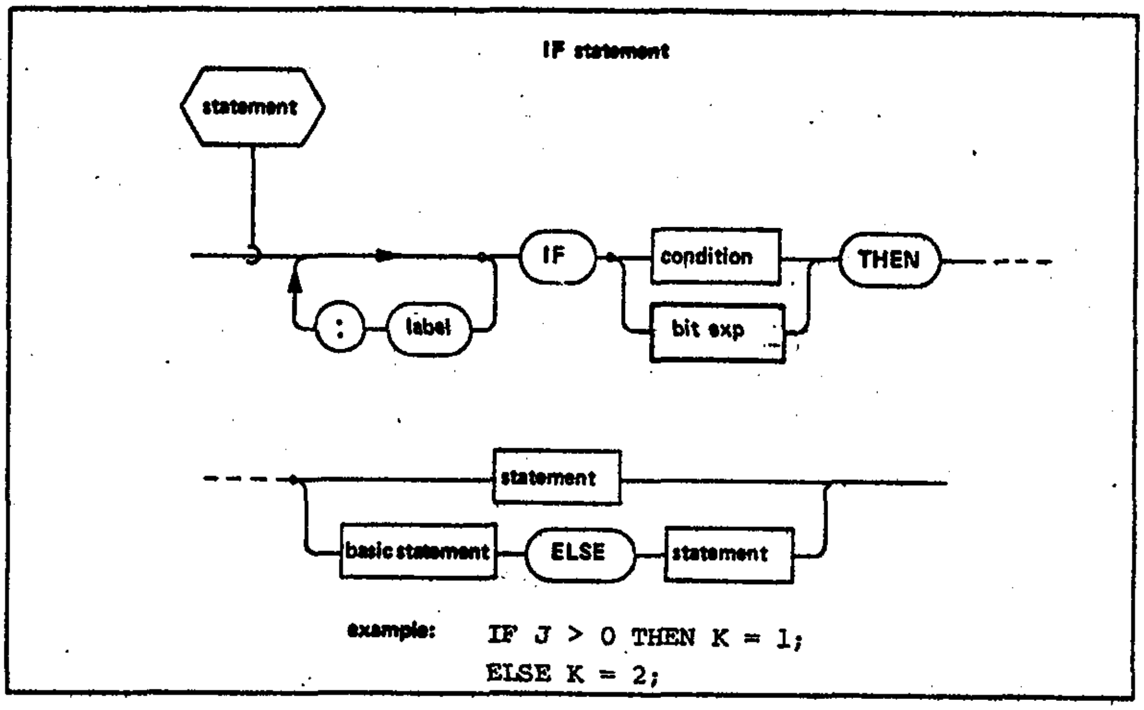


Any <basic statement>, unless it is imbedded in an IF statement or ON ERROR statement, may optionally be labeled with any number of <label> s. Not all forms of <basic statement> are described in this Section. Real time programming statements are described in Section 8, error recovery statements in Section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

Syntax



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 disallowed.

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.

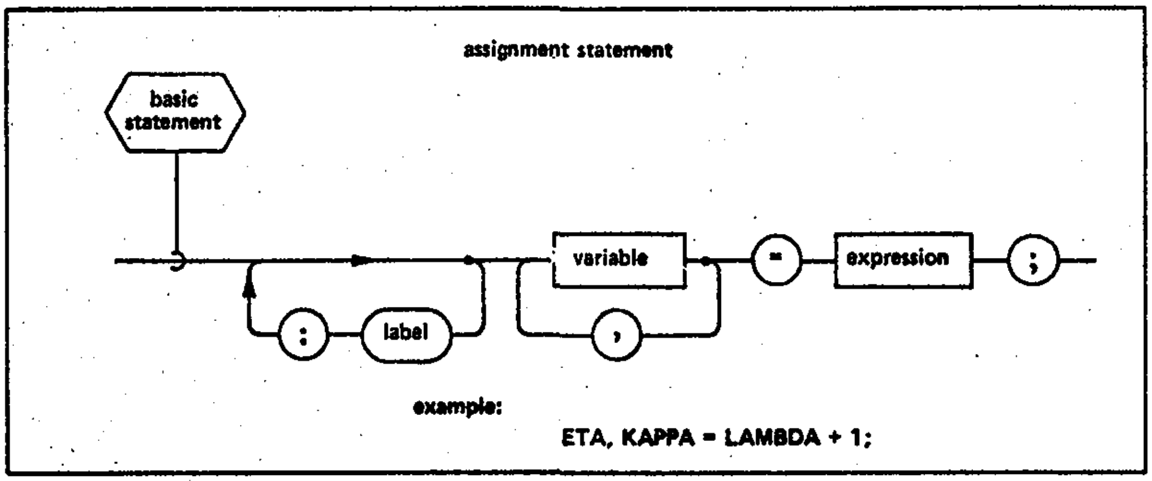
5. If the <condition> or <bit exp> is FALSE then the <statement> or <basic statement> following the keyword THEN is not executed. If the ELSE clause is present then the <statement> following the keyword ELSE is executed instead, and then execution proceeds to the <statement> following the IF statement. 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



General Semantic Rules

1. A < variable> may not be an event variable, a dcw variable or an input parameter of a procedure or function block.

2. The effective order of execution of an assignment statement is as follows:

   o any subscript expressions on the left-hand side are evaluated,

   o the right-hand side <expression> is evaluated, and

   o 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 left-hand side to be arrayed. If more than one <variable> is arrayed, the arraynesses 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 the <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:

   o Assignment of an integer <expression> to a scalar <variable> is allowed. Depending on the implementation this may cause loss of decimal places of accuracy.

   o Assignment of a scalar <expression> to an integer <variable> is allowed, causing rounding to the nearest integral value. This may cause a run time error if, in any implementation, the scalar has too large an absolute value to be represented as an integer.

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 dimension(s) 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.

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 6.6.4.

<u>Semantic Rules: Character Assignments</u>

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.

2. If <variable> is a character variable with no component subscripting, then:

   o 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.

   o 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:

   o 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.

   o 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.

   o After assignment the <variable> takes on the length implied by the upper index of the component subscript, or retains its original length, whichever 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.

   o If the lower index is greater than the length of the <variable> before assignment, then the intervening gap is filled with blanks.

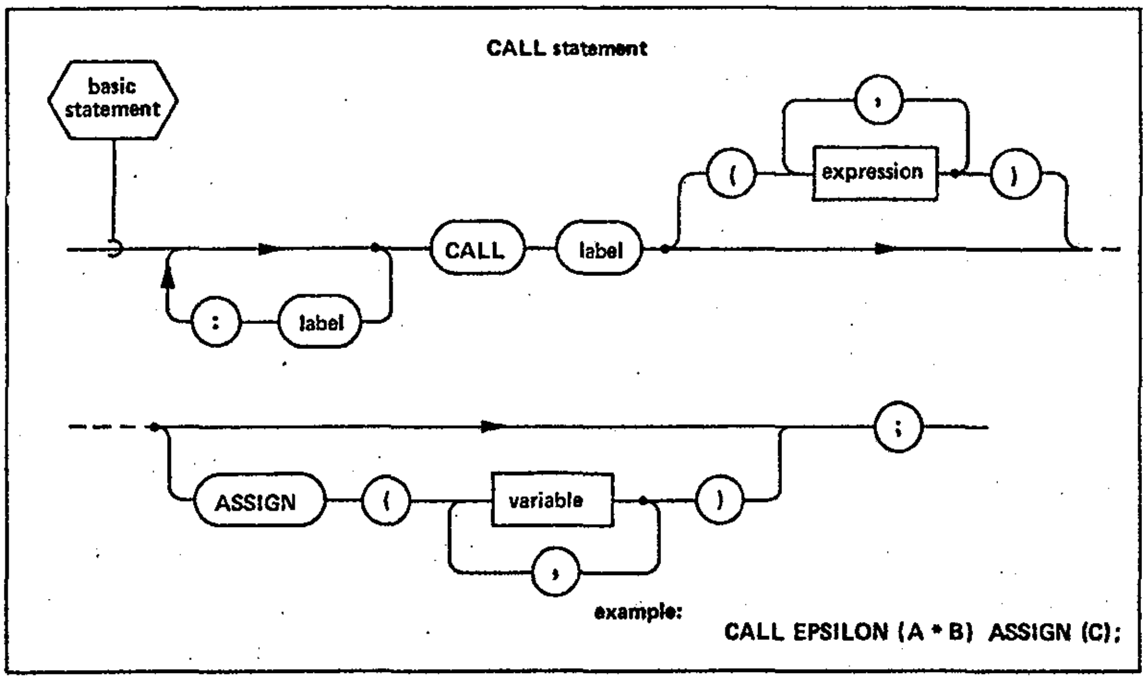
<u>Semantic Rules: Structure Assignments</u>

1. An <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



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 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).

4. Each of the < expression> s is an "input argument" of the procedure call.

5. 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:

   o No component subscripting for bit and character types.

   o If component subscripting is present, < variable> must be subscripted so as to yield a single (unarrayed) element of the <variable> .

   o If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

6. Assign arguments are "call-by-reference." Input arguments are either "call-by-reference" or "call-by-value." (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.)

7. Each assign argument must match its corresponding procedure block assign parameter exactly in type, precision, dimension, arrayness, structure tree organization, and DENSE and REMOTE attributes, as applicable. CHARACTER lengths are an exception; the declared lengths need not match. 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:

   o If it is of lock group N, then the corresponding assign parameter must be of lock group N, or *.

   o If it is of lock group *, then the corresponding parameter must also be of group *.

8. Bit type identifiers which are part of structure variables and have the DENSE attribute may not be used as ASSIGN arguments of a CALL statement. 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.

9. A <dcw var> or an <event var> may not be "assign arguments" of the procedure call.

10. For input arguments, the following relaxations of rules 7 and 8 are permitted:

- o precisions need not match,
- o the lengths of bit arguments need not match,
- o the lengths of character arguments need not match,
- o implicit integer to scalar and scalar to integer conversions are allowed,
- o implicit integer and scalar to character conversions are allowed, and
- o matching of the attributes DENSE and REMOTE is not required.

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.

Example:

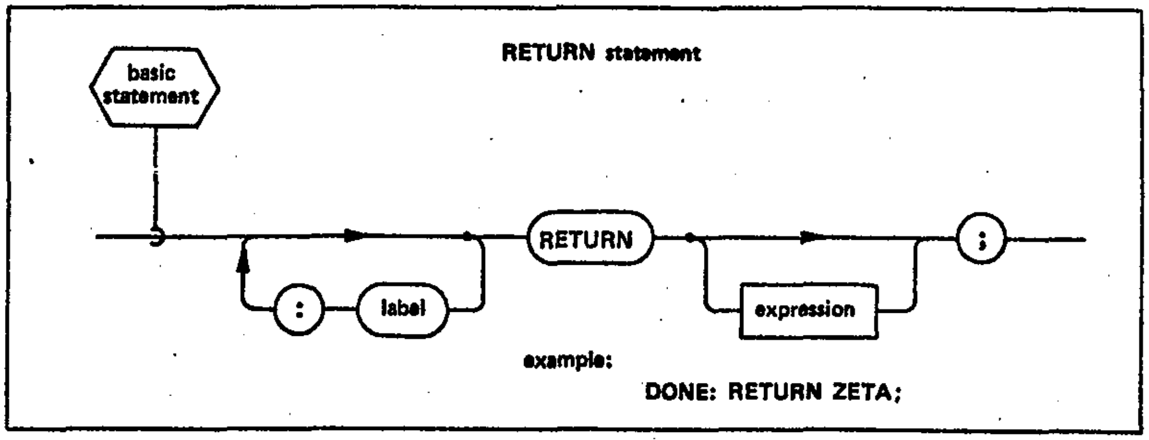
```
STRUCTURE Z:

     1 A,
        2 C CHARACTER (80)
        2 B VECTOR,
     1 D INTEGER;
DECLARE ZZ Z-STRUCTURE (20);
•
•
•
```

$$\text{CALL X ASSIGN (}\underset{\underset{\text{legal}}{\uparrow}}{\text{ZZ}}\text{, }\underbrace{\text{ZZ.A, ZZ.A.B}}_{\text{illegal}}\text{, }\underset{\underset{\text{legal}}{\uparrow}}{\text{ZZ.A}_1}\text{);}$$

7.5 The RETURN Statement

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

Syntax



General Semantic Rules

1. The effect of the RETURN statement is to cause normal exit (return of execution) from a TASK, CRITICAL, PROCEDURE, or FUNCTION block. (Also see the CLOSE statement, Section 3.8.4).

2. An <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. An <expression> must match the function definition in type and dimension, with the following exceptions:

   o the lengths of bit expressions need not match,

   o the lengths of character expressions need not match,

   o implicit integer to scalar and scalar to integer conversions are allowed, and

   o implicit integer and scalar to character conversions are allowed.

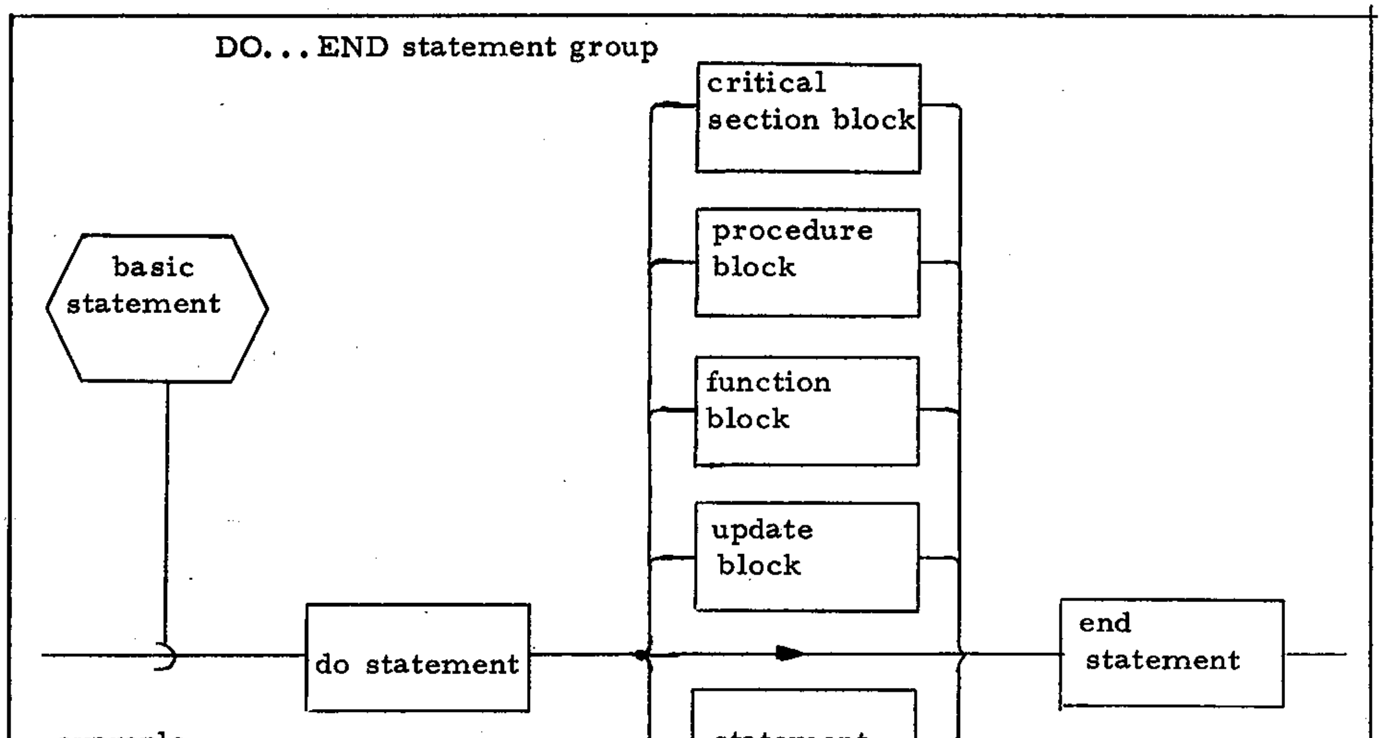
   The return of the function values may be viewed as the assignment of the < expression> to the function name. The rules applicable in the above exceptions thus parallel the relevant assignment rules given in Section 7.3.

4. An <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



example:

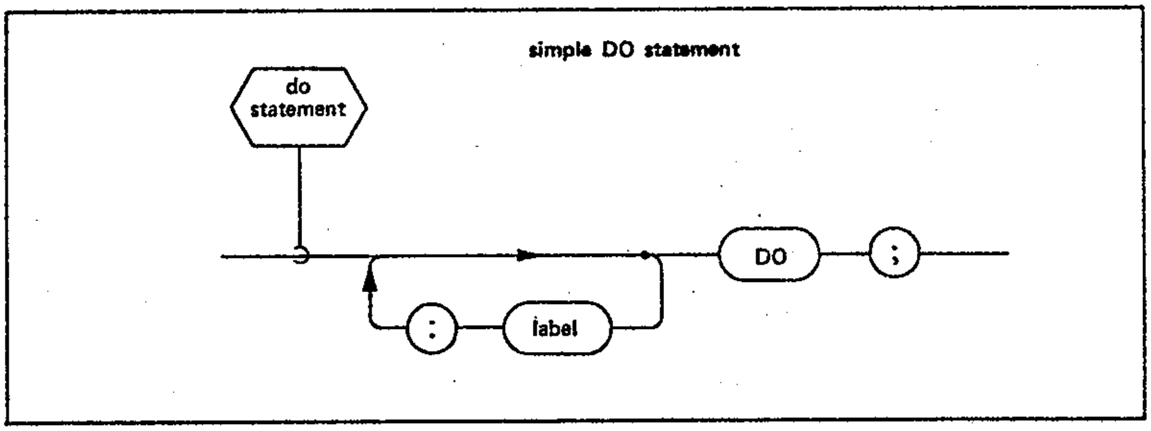
```
    DO WHILE J>0;
        J = J - 1;
    END;
```

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, CRITICAL SECTION, 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 once only.

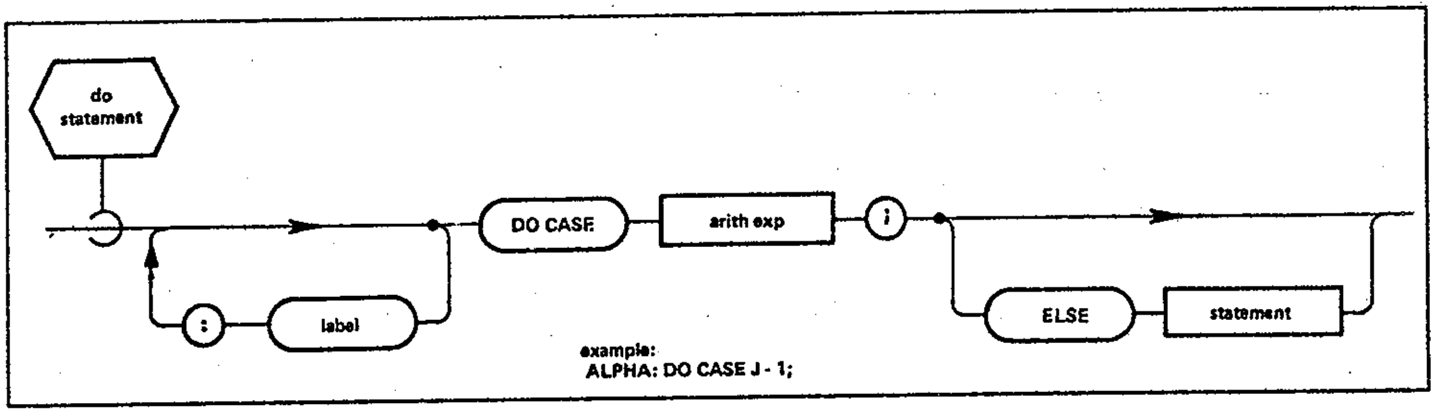
Syntax



7.6.2 The DO CASE Statement

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



Semantic Rules

1. An <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 Kth <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 a run time error occurs.

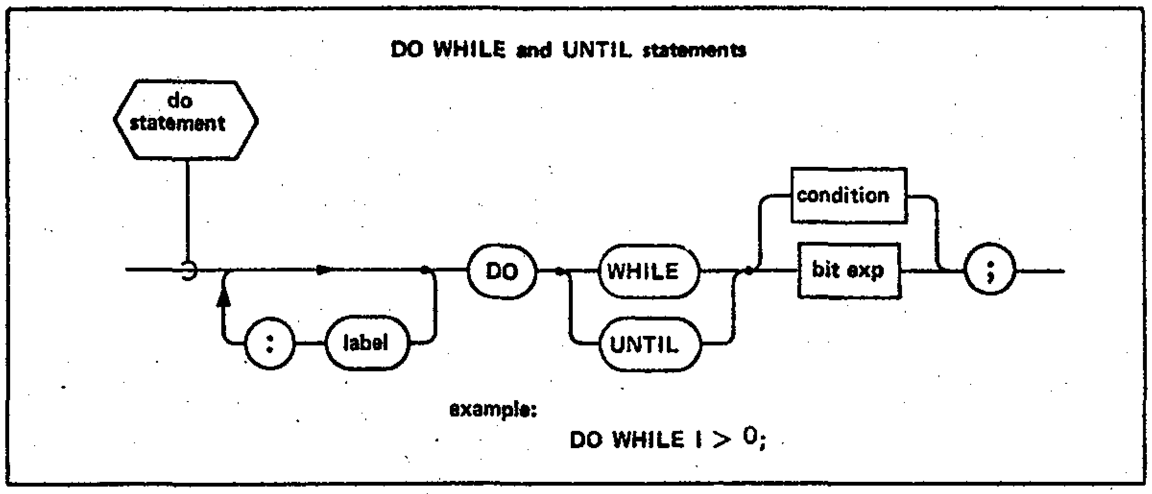
4. If the value of K is outside the range defined in Rule 2, but an ELSE clause does appear, the <statement> following the ELSE keyword is executed instead of one of those in the group. The option to label <statement> is disallowed.

5. The presence of any code block definition in the group of <statement> s does not change the K-indexing of the <statement> s.

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



Semantic Rules

1. There is no semantic restriction on <condition> . A <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.

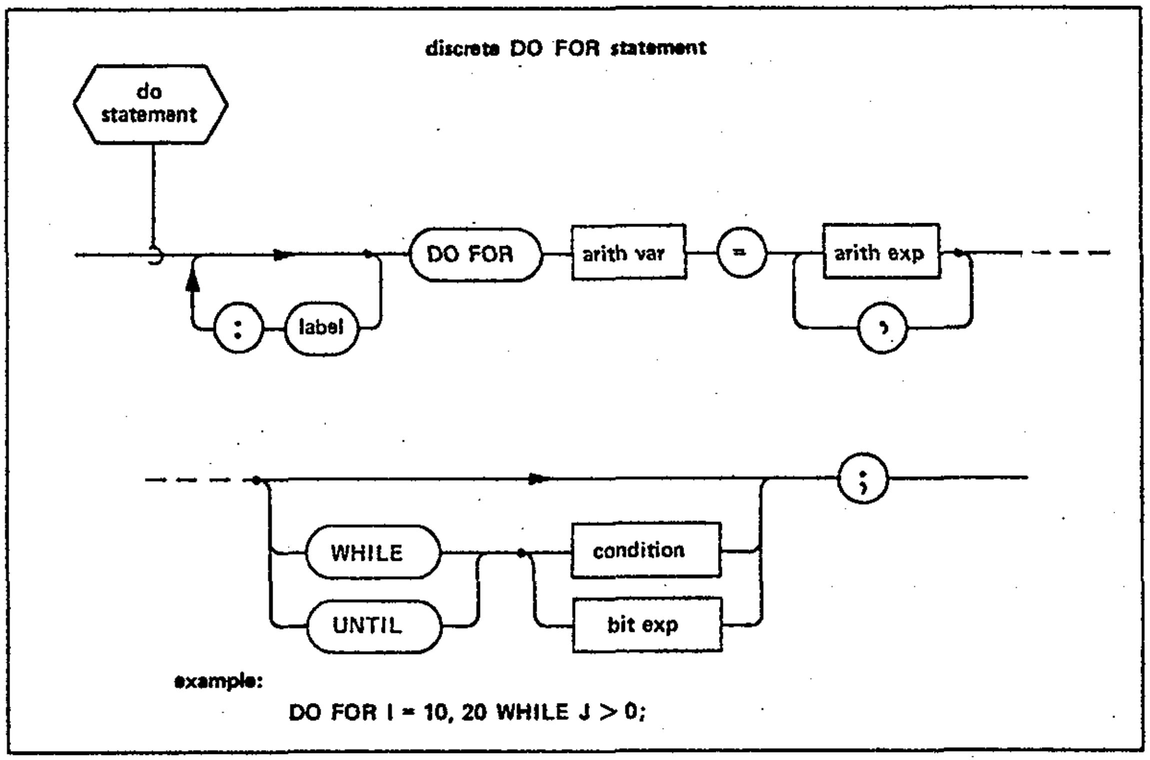
2. In the DO WHILE version, the group of <statement> s is repeatedly executed until the value of <condition> or <bit exp> becomes FALSE. The value is tested at the beginning of each cycle of execution. This implies that if <condition> or <bit exp> is initially FALSE the group of <statement> s is not executed at all.

3. In the DO UNTIL version, the group of <statement> s is repeatedly executed until the value of <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



Semantic Rules

1. An <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable.

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. An <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 on <condition>. A <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.

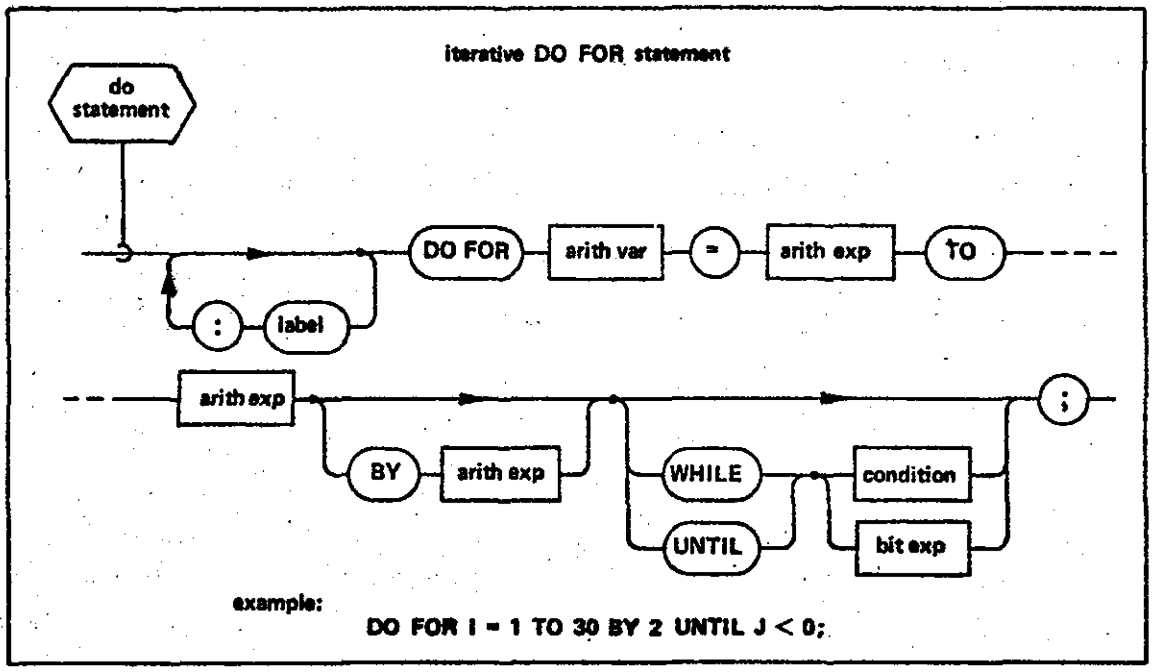
7. If the WHILE clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes FALSE. The value is tested at the beginning of each cycle of execution after the assignment of the loop variable. This implies that if <condition> or <bit exp> is FALSE prior to the first 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



Semantic Rules

1. An <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable.

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. Unless a BY clause appears in the DO FOR statement, the value assigned to the loop variable prior to the Kth cycle of execution is one greater than its value on the K-1th cycle.

4. If a BY clause appears in the DO FOR statement, the value assigned to the loop variable prior to the Kth cycle of execution is equal to its value on the K-1th cycle plus the value of <arith exp> following the BY keyword (the "increment").

5. 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.

6. 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").

7. 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.

8. 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.

9. 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.

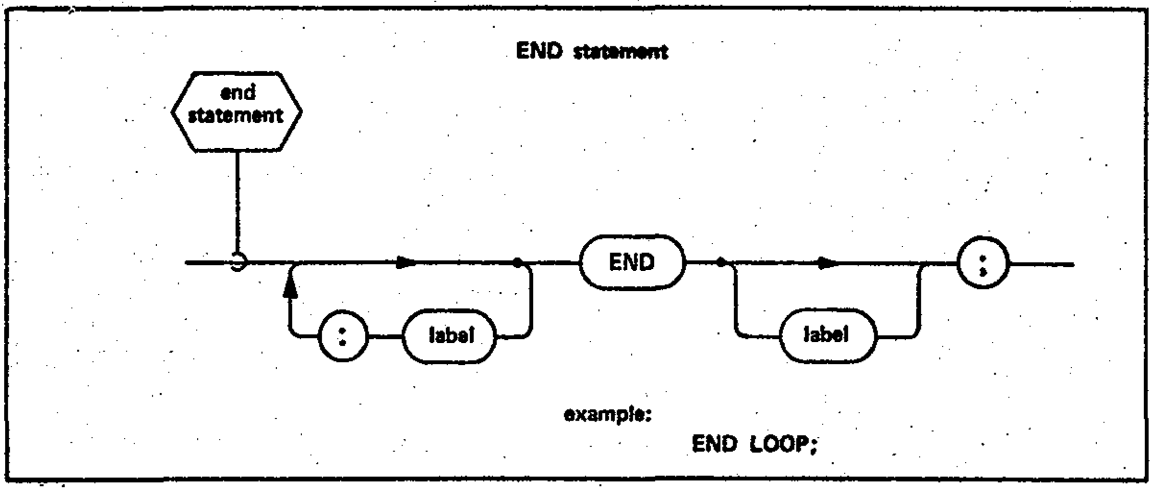
10. If the UNTIL clause is used, cycling of execution is abandoned when the value of <condition> or <bit exp> becomes TRUE. The value is not tested before the first cycle of execution. On the second and all

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



Semantic Rules

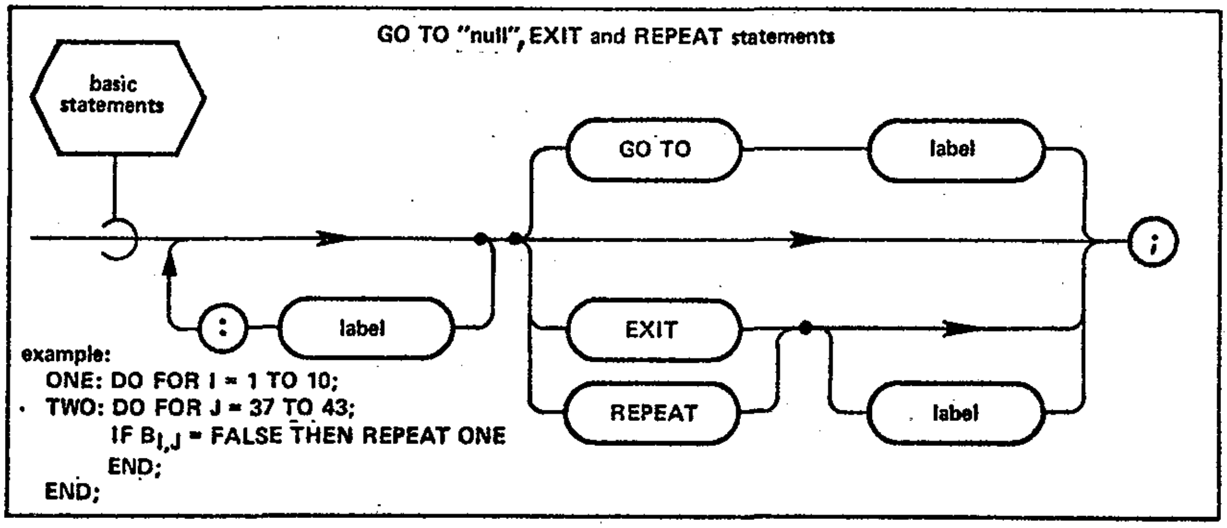
1. If <label> follows the END keyword, then it must match a <label> on the <do statement> opening the DO...END group.

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.

7.7 Other Basic Statements

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

Syntax


GO TO "null", EXIT and REPEAT statements
basic statements
GO TO
label
:
label
EXIT
REPEAT
label
;
example:
ONE: DO FOR I = 1 TO 10;
TWO: DO FOR J = 37 TO 43;
IF B_{I,J} = FALSE THEN REPEAT ONE
END;
END;

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 the 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 run time.

3. The EXIT statement is legal only within a DO...END group or within nested such groups. 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 legal only within a DO...END group opened with a DO FOR, DO WHILE, or DO UNTIL statement, or within nested such groups. 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 conditions of the opening <DO statement> are still satisfied, the next cycle of execution begins normally.

5. Code blocks (procedures, 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.

(BLANK)

8. REAL TIME CONTROL

HAL/SM contains a comprehensive facility for creating a multi-tasking job structure in a real time programming environment. MOSS controls loading and initiation of jobs and at run time controls the execution of tasks held in a task queue. HAL/SM contains statements which load and initiate jobs, schedule tasks (enter them in the task queue), terminate jobs and tasks (remove them from the task queue), and otherwise direct MOSS in its controlling function. HAL/SM also contains means whereby the use of data or resources by more than one task at a time is managed in a safe, protected manner at specific localized points within the tasks.

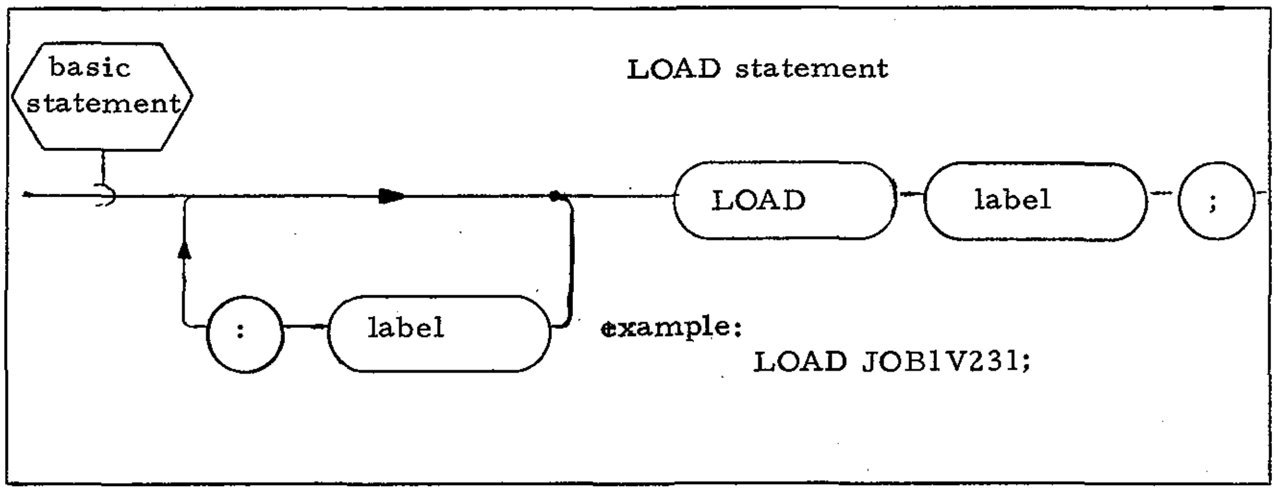
8.1 Jobs and MOSS

Under MOSS a job may be linked, loaded, and/or initiated through the use of MOSS Job Control Language. Also, an executing job may request that another job be loaded, initiated, and/or terminated. In HAL/SM the LOAD statement, the INITIATE statement, and the TERMINATE statement specify the job conditions mentioned above.

8.1.1 The LOAD Statement

The LOAD statement specifies that a particular job should be loaded from the Job Load Library to External Paging Memory under MOSS control.

Syntax



Semantic Rules

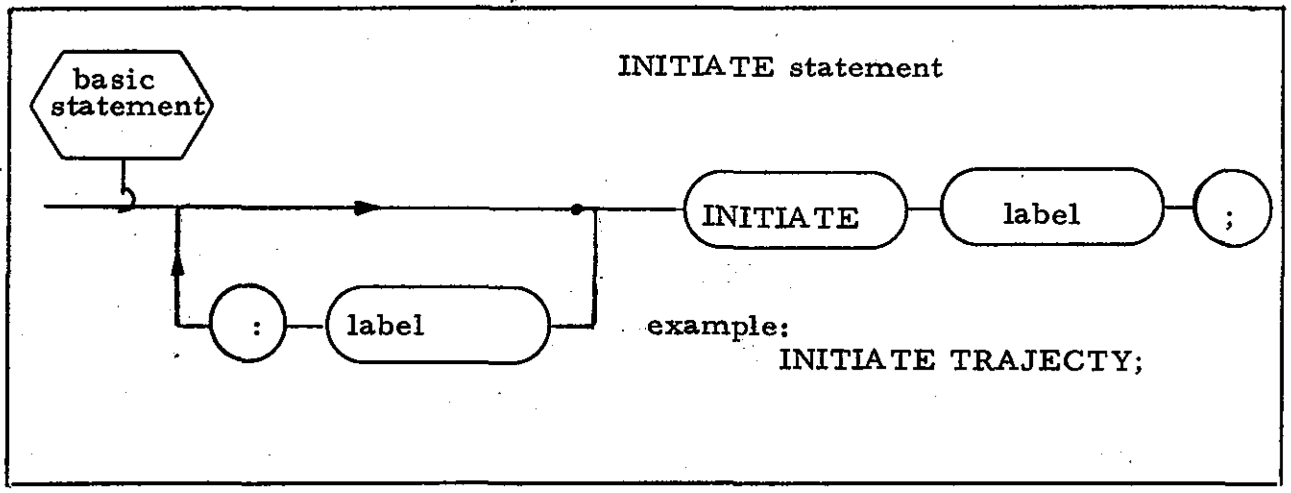
1. The <label> following LOAD must be limited to eight characters and must be declared with the JOB label attribute.

2. The < label> following LOAD must identically match the job name of a job in the MOSS Job Directory

8.1.2 The INITIATE Statement

The INITIATE statement specifies MOSS execution of a particular job that has been previously loaded.

Syntax



Semantic Rules

1. The < label> following INITIATE must be limited to eight characters and must be declared with the JOB label attribute.

2. The < label > following INITIATE must identically match the job name of a job in the MOSS Job Directory.

8.2 Tasks and MOSS

In HAL/SM, a task may be scheduled for execution and placed in the task queue. Each task in the MOSS task queue is at any instant in one of a number of states. For the purposes of this section, the following states are defined: (These states are not necessarily definitive of those actually existing in MOSS.)

- Active - a task is said to be in the active state if it is actually in execution. Depending on the implementation it may be possible for several tasks to be in execution simultaneously.
- Wait - a task is said to be in the wait state if it is ready for execution but MOSS has decided on a priority basis that its execution should be delayed or suspended.
- Ready - a task is said to be in the ready state if it is in either the active or the wait states.
- Stall - a task 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 task being brought into the active state for the first time is called "initiation."

Execution of a CLOSE or RETURN statement at the task level, or a TERMINATE task statement at any level causes termination of the task and return to MOSS. Execution of a TERMINATE job statement causes termination of the task and the job when all other tasks in the job have terminated.

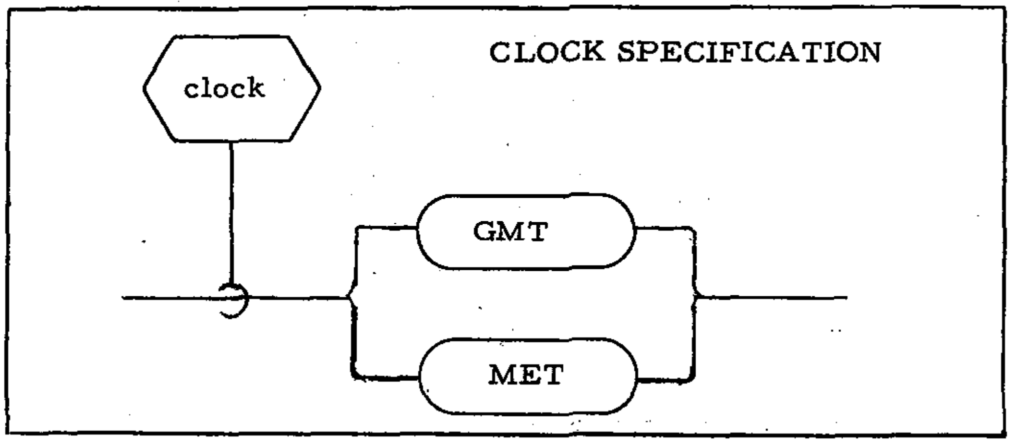
8.3 Timing Considerations

In the HAL/SM system, MOSS contains clocks measuring elapsed time. Time is measured in milliseconds. HAL/SM contains several instances of timing expressions which in effect make reference to these clocks.

Absolute time may be expressed relative to one of two time scales, Greenwich Mean Time (GMT) or Mission Elapsed Time (MET). Alternatively, time may often be expressed relative to the present time, depending on the syntax of a specific statement.

The syntactical term <clock> may be used to specify either Greenwich Mean Time or Mission Elapsed Time.

Syntax



8.4 The SCHEDULE Statement

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

- A task may be scheduled so that MOSS immediately places it in a ready state.

- Conditional execution of a task may be specified so that a task will be required to wait for certain specified conditions to be satisfied before the task can be considered ready for execution. Such conditions include:

  - single or multiple events combined in a logical expression,

  - a specified period of elapsed time,

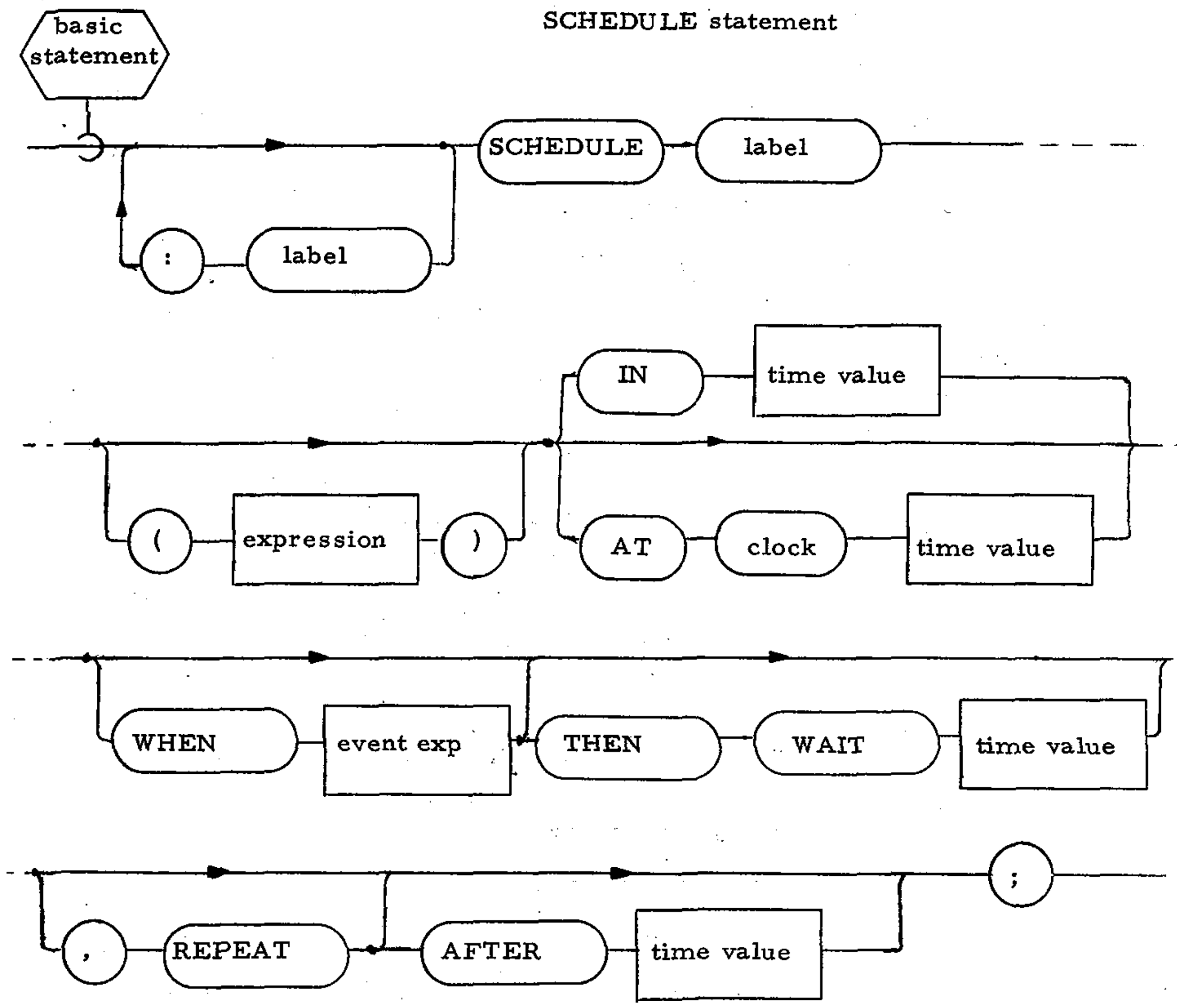
  - a clock time,

  - combinations of the above.

  Combinations may specify either that the events are not to be considered until after some period of elapsed time or until after some clock time, or that a period of time must elapse after the events are satisfied.

- Up to 256 bytes of data may be passed to the scheduled task.

Syntax



example:
SCHEDULE IOTA;
SCHEDULE DELTA (PARMS) AT GMT 31000, REPEAT AFTER 50 MINS;

Semantic Rules

1. SCHEDULE< label> schedules a task with the name <label >, as defined in the JCL task statement, placing a new task with name < label> in the task queue. Unless otherwise specified, MOSS puts the new task in the ready state immediately after execution of the SCHEDULE statement.

2. The phrase IN< time value> is used to cause the task to be put in the stall state for a fixed time duration. The< time value> is evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then the task is put immediately in the ready state if no other conditions are specified.

3. The phrase AT< clock> < time value >is used to cause the task to be put in the stall state until a fixed clock (GMT or MET) time. The < time value >is evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than the current clock time, then the task is put immediately in the ready state unless otherwise specified.

4. The phrase WHEN< event exp> is used to cause the task to be put in the stall state until some event condition is satisfied. Starting from the time of execution of the SCHEDULE statement (or after the time delay specified in an AT or IN phrase (if specified)), the< event exp> is evaluated at each "event change point" (see Section 8.10) until its value becomes TRUE. At that time the task is placed in the ready state unless otherwise specified. If the value of< event exp> is TRUE upon execution of the SCHEDULE statement, then the task is immediately put in the ready state unless otherwise specified.

5. The phrase THEN WAIT <time value> is used to cause the task to be put in the stall state for a fixed clock duration following the satisfaction of any previous conditions. The< time value> is evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then the task is put immediately in the ready state.

6. The SCHEDULE statement has no effect on the specified task and an error is issued if that task has already begun execution, or in the case of a periodic task, has not been cancelled. However, if a conditional SCHEDULE statement has been previously issued for a task and has not been satisfied), then the current SCHEDULE command replaces the previous one.

7. A task cannot transmit more data than can be contained in the area specified (to a maximum of 256 bytes) via the JCL for the receiving task. If too much data is specified, the receiving task is not initiated and an error is issued to the sending task.

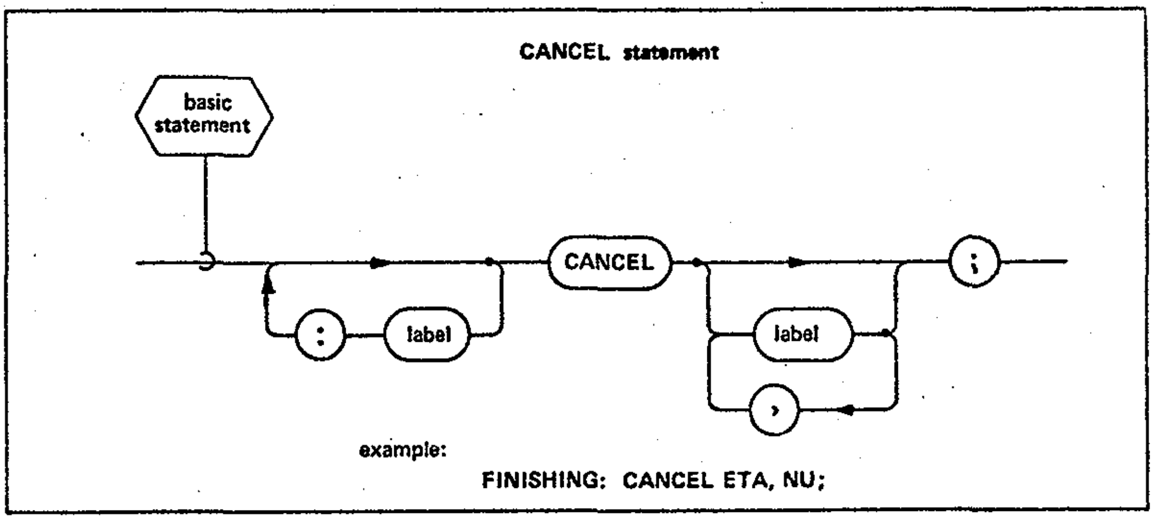
8. When a conditional SCHEDULE command is issued for a periodic task, the non-periodic conditions are required to be satisfied prior to the first execution only. Subsequent scheduling of the task is based on the <time value> specified in the AFTER clause.

9. The REPEAT phrase of the SCHEDULE statement is used to specify a task which is to be executed cyclically by MOSS. If the REPEAT phrase is not qualified, then cycles of execution follow each other with no intervening time delay. To cause execution of consecutive cycles to be separated by a fixed intervening time delay, the qualifier AFTER <time value> is used. The <time value> is evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then no time delay results.

8.5 The CANCEL Statement

Cancellation of a task implies the removal of the specified task from the task queue upon termination. Cancellation may only be the result of executing a CANCEL statement.

Syntax



Semantic Rules

1. CANCEL< label> causes cancellation of the task <label> . The CANCEL statement can be used to cancel any number of tasks simultaneously.

2. If the CANCEL statement has no <label> , cancellation of the task executing the CANCEL statement is implied.

3. If at the time of execution of the CANCEL statement, a task to be cancelled has not yet been initiated, then the task is merely removed from the task queue. This applies to both cyclic and non-cyclic tasks.

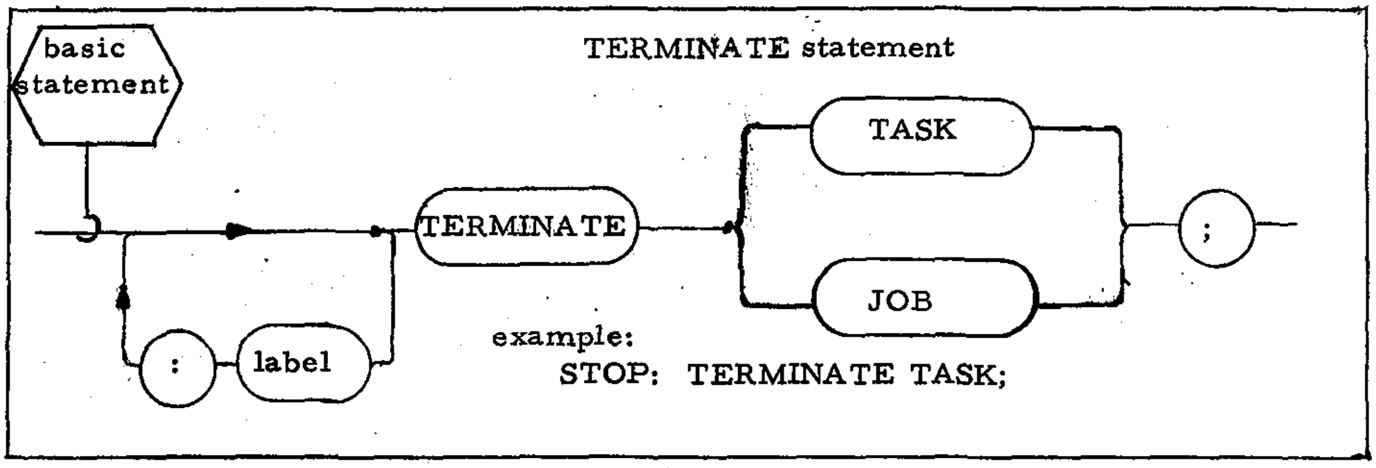
4. If at the time of execution of the CANCEL statement, a task to be cancelled has already been initiated, then the following ensues. If the task is non-cyclic and it has already been initiated, the CANCEL statement has no effect. If the task is cyclic, then the task is cancelled at the end of the current cycle of execution.

5. If at the time of execution of the CANCEL statement, the specified task is not in the proper state to be cancelled, a runtime error will result.

8.6 The TERMINATE Statement

This statement allows the user to signal the completion of execution to MOSS of the currently executing task and optionally signal completion of execution of the job to which it belongs.

Syntax



Semantic Rules

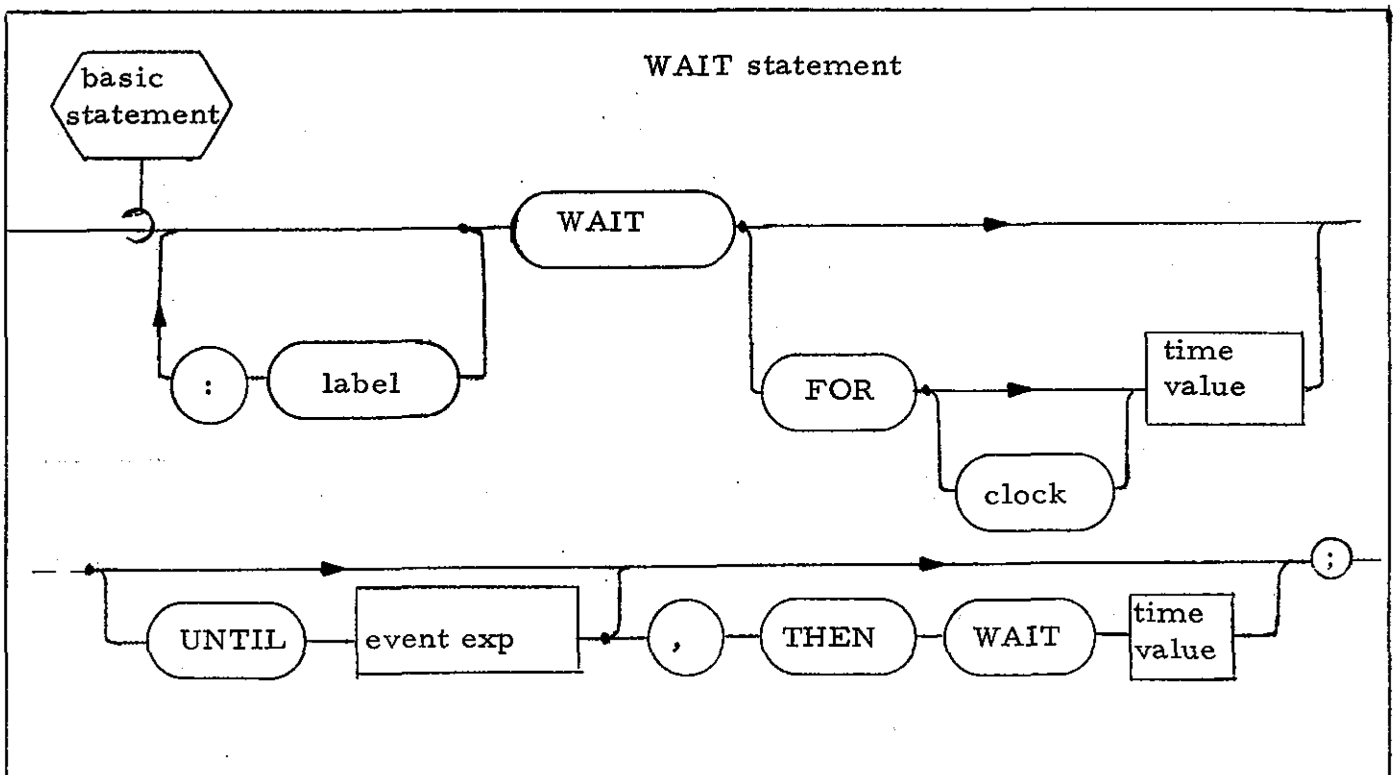
1. TERMINATE TASK causes the termination of the task executing the TERMINATE statement.

2. TERMINATE JOB implies task termination. Other tasks within the job are allowed to proceed to completion. No new task belonging to the job associated with this task is permitted to begin execution.

8.7 The WAIT Statement

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

Syntax



example:
NOW: WAIT UNTIL EVENT_A & EVENT_B, THEN WAIT 30 MSECS;

Semantic Rules

1. The WAIT FOR< time value >version specifies that the task executing the WAIT statement is to be placed in the stall state for a clock duration fixed by the value of the< time value>. The< time value> is 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 FOR < clock> < time value> version specifies that the task executing the WAIT statement is to be placed in the stall state until a clock time fixed by the value of the< time value>. The< time value> is evaluated once at the time of execution of the WAIT statement. If the value is not greater than the current clock time, the WAIT statement has no effect.

3. The WAIT UNTIL <event exp> version specifies that the task 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, or after the time delay specified in a FOR phrase (if specified), the <event exp> is evaluated at every "event change point" until its value becomes TRUE, whereupon the task is returned to the READY state if no THEN WAIT<time value> phrase is specified. If the value of <event exp> is TRUE upon evaluation, then the phrase has no effect.

4. The phrase THEN WAIT<time value> is used to cause the task to be put in the stall state for a fixed clock duration following the satisfaction of any previous conditions. The <time value> is evaluated once at the time of execution of the WAIT statement. If the value is not greater than zero then the process is put immediately in the ready state.

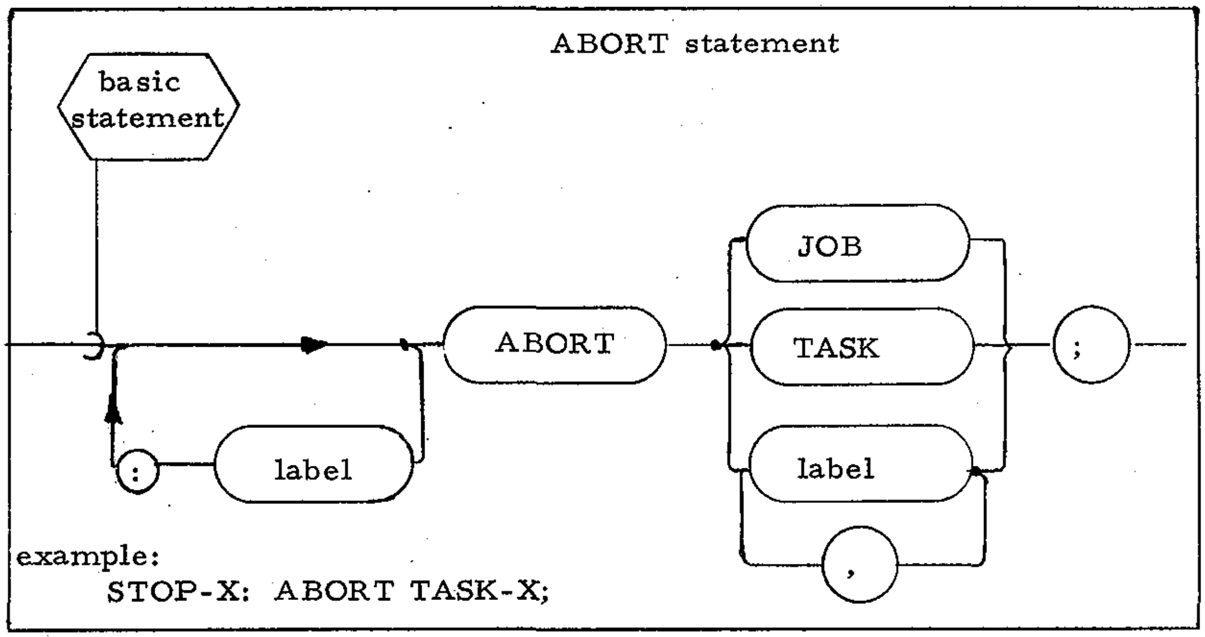
5. Although each of the phrases of this statement are optional, at least one must be specified.

6. When a task suspends its execution with the WAIT statement, all resources that it has selected are automatically released (see Section 8.15).

8.8 The ABORT Statement

The ABORT statement allows the user to cause any task in the same job to be aborted, or the entire job to which the task belongs to be aborted.

Syntax



Semantic Rules

1. The ABORT JOB version specifies that the task executing the statement and all other tasks within the job are immediately aborted.

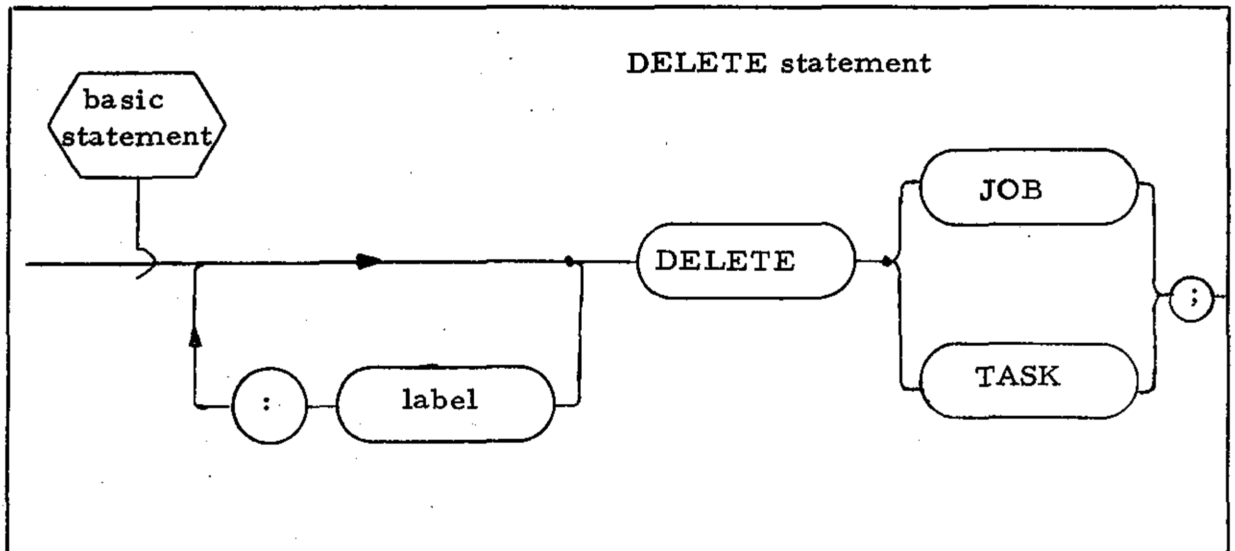
2. The ABORT TASK version specifies that the task executing the statement is to be immediately aborted.

3. The ABORT <label> version specifies one or more tasks within the same job are to be aborted. The <label> must be the name of a task as defined in the Task statement in the JCL.

8.9 The DELETE Statement

The DELETE statement provides a means through which a task may terminate its execution and release its Main Memory or terminate the job to which it belongs and remove it from EPM.

Syntax



example:
 DELETE JOB;

Semantic Rules

1. The DELETE TASK form will cancel a periodic task.

2. DELETE TASK also implies task terminate and all of the actions performed at task termination will be executed for DELETE TASK.

8.10 Event Control

Although a formal specification of event variables, flags, and event expressions has already been given 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. However, the user may not directly assign a value to an event variable; this can only be done by MOSS in response to the occurrence of a particular event which the event variable is monitoring, or in response to the execution of a RESET statement referencing the event variable.

Event variables are used to monitor the activity of events. The particular event that is being monitored at any given point in time is assigned to an event variable via the ALERT statement. Unassigned (unalerted) event variables or alerted event variables for which the event has not occurred have a value of zero (or FALSE or OFF). The value of the event variable is set to one (TRUE or ON) by MOSS when the event occurs.

There are three basic categories of events:

- o Task termination
- o Program flags
- o RTIOS data bus state changes

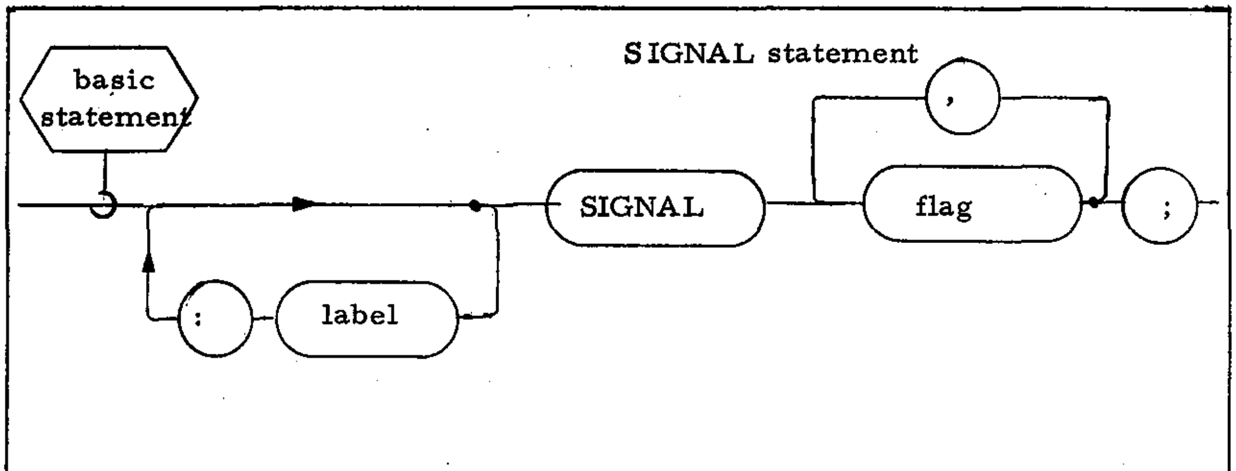
The concept of a task termination event is self-explanatory. Data bus state changes will be described with the ALERT statement. Program flags are basically user defined events which may be set (i.e., caused to occur) when a task executes a SIGNAL statement referencing the particular program flag(s) which the programmer wishes to use to indicate a particular situation has occurred. This allows the programmer to use the event mechanism as a form of inter-task communication and synchronization via the SCHEDULE and WAIT statements.

When an event occurs, any event variables which happen to be alerted to that event are set to one (any number of event variables may be alerted to an event at any given time). The occurrence of the event removes the assignment of the event variable(s) to the event; however, the event variable retains the ON value until the event variable is realerted to the same or another event, or until the event variable is explicitly RESET by the programmer.

8.11 The SIGNAL Statement

The SIGNAL statement permits the programmer to "cause" the occurrence of a programmer defined event by signaling a program flag. Any event variables alerted to the flag variable will be set to one (or TRUE or ON).

Syntax



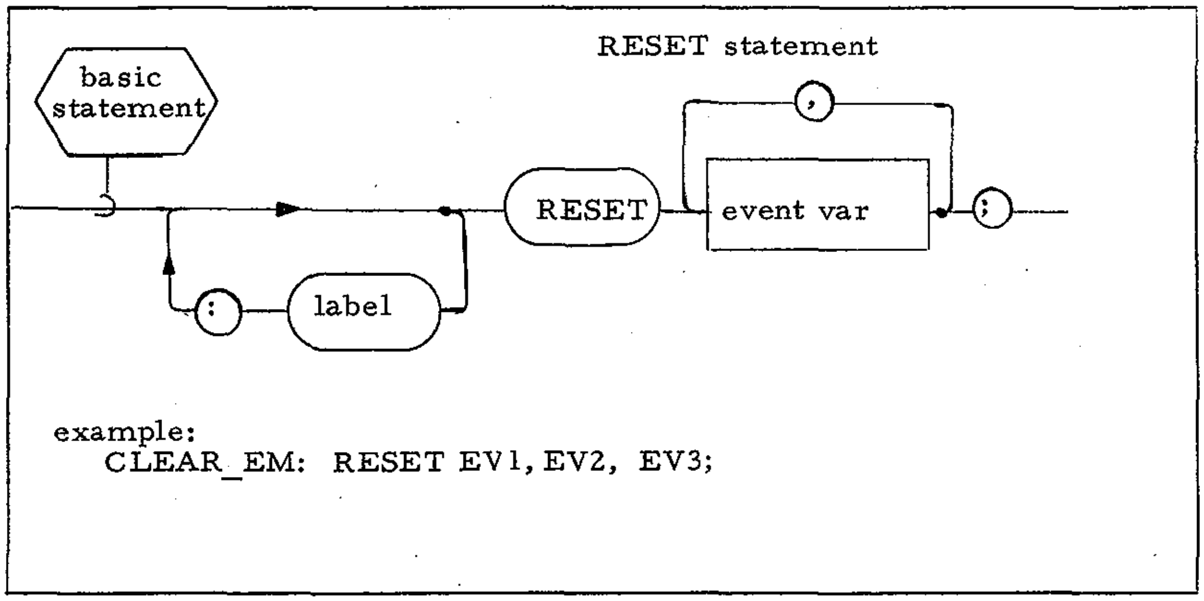
example:

```
    L:  SIGNAL BETA, IOTA;
```

8.12 The RESET Statement

The RESET statement causes the value of an event variable to be set to zero.

Syntax



Semantic Rules

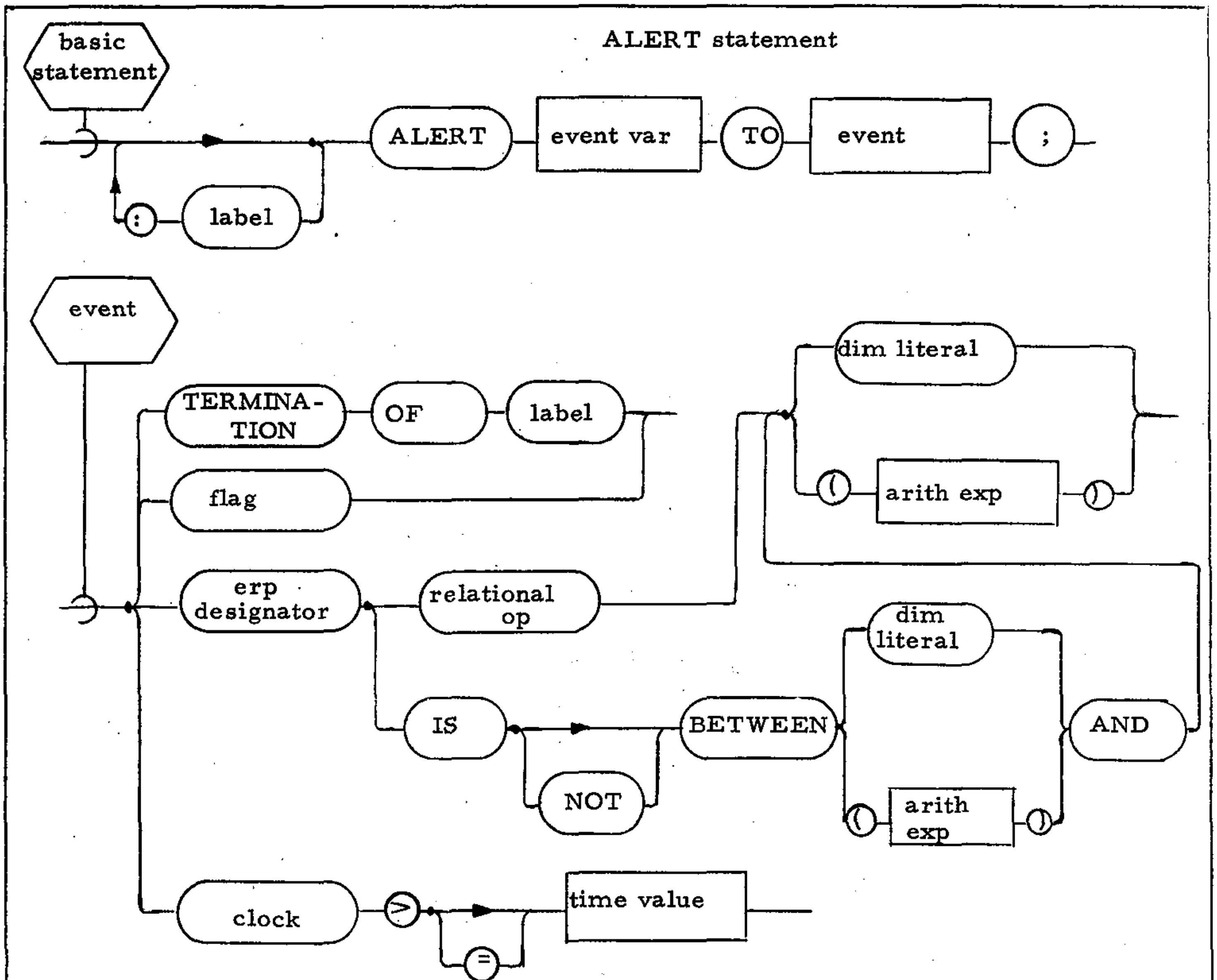
1. The value of an event variable is set to zero, independent of whether the event variable was previously one or zero, or whether it had been previously alerted or not.

2. If an event variable has been alerted to an event and then RESET before the occurrence of the event, the alert is no longer in affect (i.e., MOSS ceases to associate the occurrence of the event with the event variable).

8.13 The ALERT Statement and Events

The ALERT statement enables a task to specify that a particular event is to be monitored and its occurrence recorded in a particular event variable.

Syntax



example:

ALERT ALPHA TO TERMINATION OF TASK_A;

Semantic Rules

1. Unassigned (unalerted) event variables or event variables for which the event has not occurred have a value of zero. When the event to which an event variable is alerted occurs, the event variable is set to a value of one.

2. The occurrence of an event removes the assignment of all event variables which are alerted to it (the value of the event variable remains one until the event variable is either realerted or explicitly RESET).

3. The <event> which is TERMINATION of <label> is signaled by MOSS when the task named <label> terminates.

4. The <event> which is <flag> occurs when a task SIGNALs that particular program flag.

5. RTIOS data bus state changes are signaled by MOSS when the indicated <erp designator> or <clock> meets the conditions indicated.

6. The kinds of conditions which can be specified as <event> s for RTIOS data bus state changes depend on the type of the <erp designator> or <clock> and is implementation dependent.

8.14 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 HAL/SM tasks. 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 tasks both require variables from the same lock group to be changed, then the first task entering its UPDATE block must complete execution of the block before the other task can enter its own UPDATE block. The second task is placed in a stall state for the duration.

2. If one task 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 task requires use of a variable from every lock group.

3. If only one of the tasks 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 tasks'UPDATE blocks is allowed. The nature of such overlap must be such as to provide exclusive use of the lock group by the task requiring its change between the point where the variable is changed and the close of the UPDATE block.

4. If both tasks only require a variable of the same lock group accessed, then execution of the two tasks' 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 tasks, or more than one lock group, then the most restrictive of Rules 1 through 4 required is applied to resolve the conflicts.

8.15 Resource Access Control

MOSS allocates resources on a job basis according to each job's requirements as specified in MOSS Job Control Language. Once allocated to a job, a resource is accessible to all tasks within the job, but only in a controlled manner. Access to a given resource must be explicitly requested before use by a task to insure against conflicts among tasks in using the resource. Resources for which access must be requested include the following:

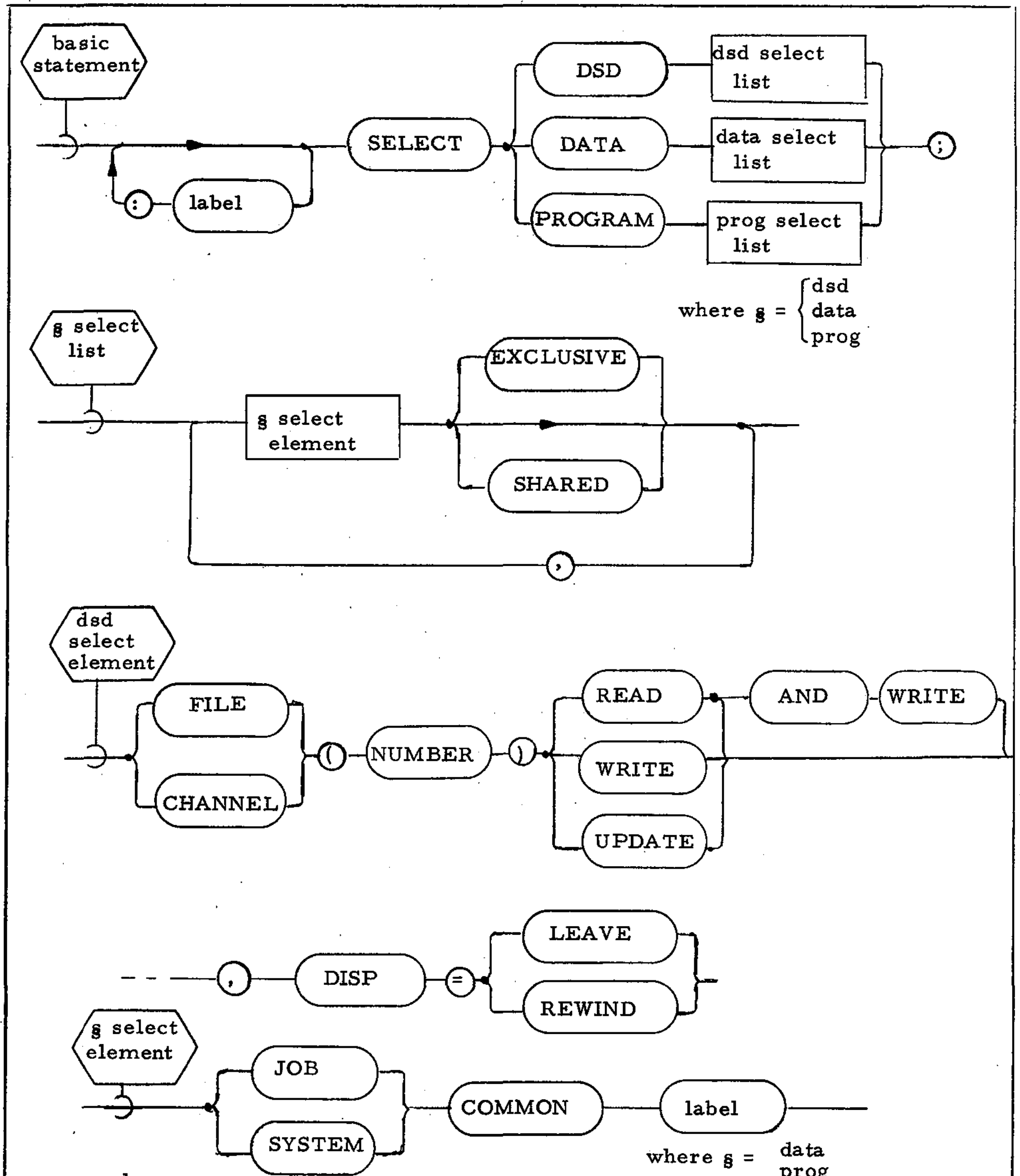
- o channels,
- o system/job common data modules (one or more COMPOOLS), and
- o system/job common serially reusable program modules (one or more COMSUBS).

There are two statements in HAL/SM, the SELECT and RELEASE statements, which specify this resource control.

8.15.1 The SELECT Statement

The SELECT statement must be used by a task to gain access to a resource before the resource may be used.

Syntax



example:

SELECT FILE CHANNEL (2) UPDATE, DISP = LEAVE;

Semantic Rules

1. The term DATA specifies a MOSS system or job common data module. This module may be one or more linked HAL/SM COMPOOLS identified by the COMPOOL <label> associated with one of them.

2. The term PROGRAM specifies a MOSS system or job common serially reusable program module. The module may be one or more linked HAL/SM COMSUBS identified by the COMSUB <label> associated with one of them.

3. The <label> following COMMON must specify the name of the data or program module. The <label> must have been declared as a COMPOOL or COMSUB name.

4. Requests for access to additional units of a particular type of resource already held by a task are rejected. All previously granted access rights to units of a particular resource type must first be released, and a single new request for all required units of the resource type issued.

5. Requests for access to resources of different types must be issued in the following order or a run time error will occur. (All previously granted access rights to resource types of a higher or equal order number must first be released before issuing a request for access rights to units of a particular resource type.)

   1 - channels and files

   2 - system/job common data modules

   3 - system/job common serially reusable program modules

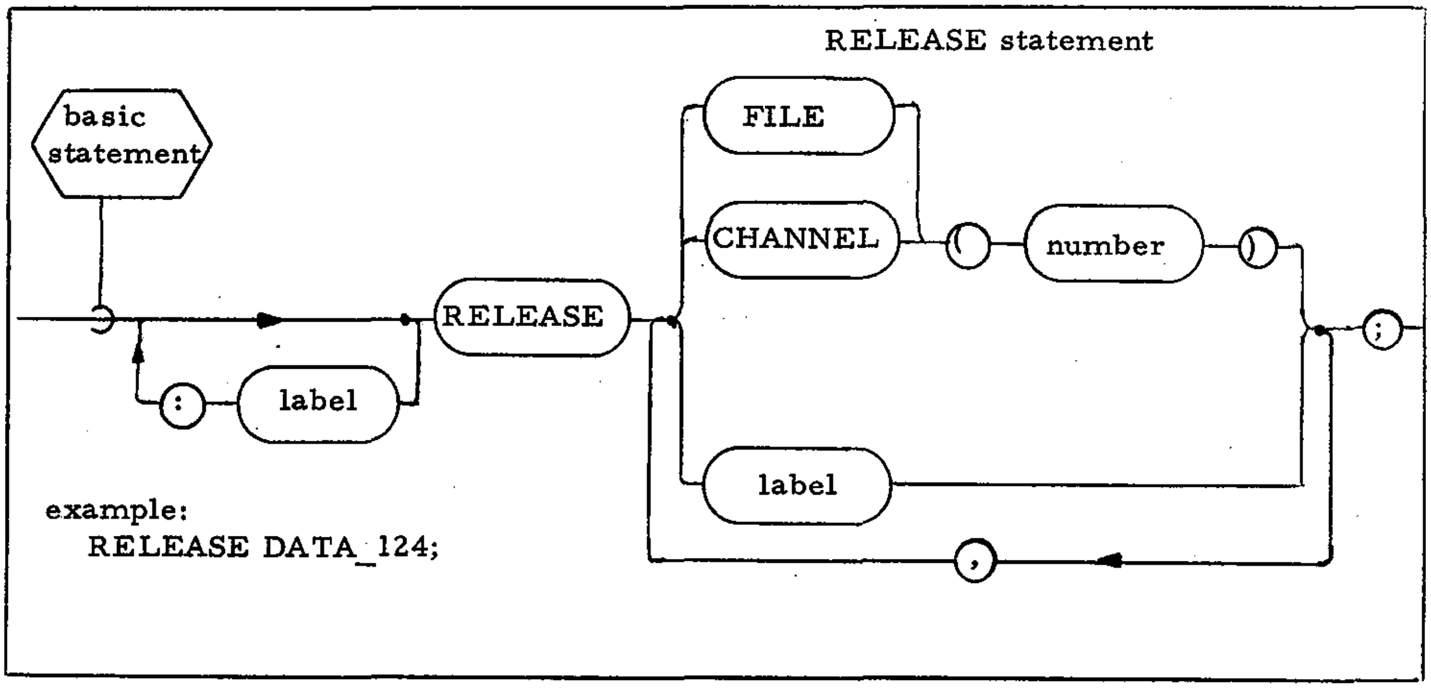
6. All resources which are SELECTed by a task are released upon entry to an UPDATE block (see Section 8.14).

7. If neither EXCLUSIVE or SHARED is specified, EXCLUSIVE shall be assumed.

8.15.2 The RELEASE Statement

The RELEASE *statement* may be used by a *task* to release access of a resource.

Syntax



Semantic Rules

1. The < label> following RELEASE must specify the name of a data or program module to be released. The < label> must have been declared as a COMPOOL or COMSUB name.

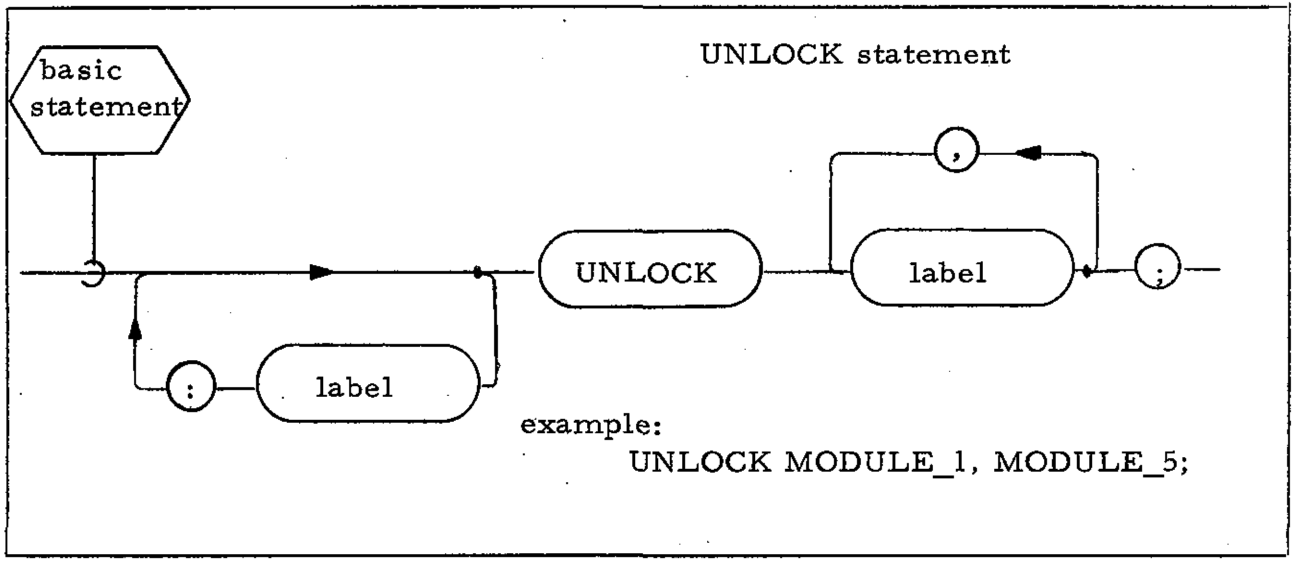
2. Releasing a CHANNEL or FILE does not "close" the channel or file.

8.16 The UNLOCK Statement

A load module may be locked in Main Memory under MOSS by using the prepaging option in MOSS JCL.

The UNLOCK statement in HAL/SM provides a means for unlocking a locked module, permitting its contents to become dynamically pageable.

Syntax



example:
UNLOCK MODULE_1, MODULE_5;

Semantic Rules

1. The < label> must reference a module that is allocated to the job of the calling task.

2. The < label > must have been identified as a COMPOOL, PROCEDURE, FUNCTION, or COMSUB name.

9. 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/SM 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 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 on 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. For system-defined errors, an implementation may define restrictions on the possible actions which the programmer may specify in lieu of the standard system action for certain errors.

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

HAL/SM possesses two error recovery and control statements. The ON ERROR statement is used to modify the error environment of a task 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.

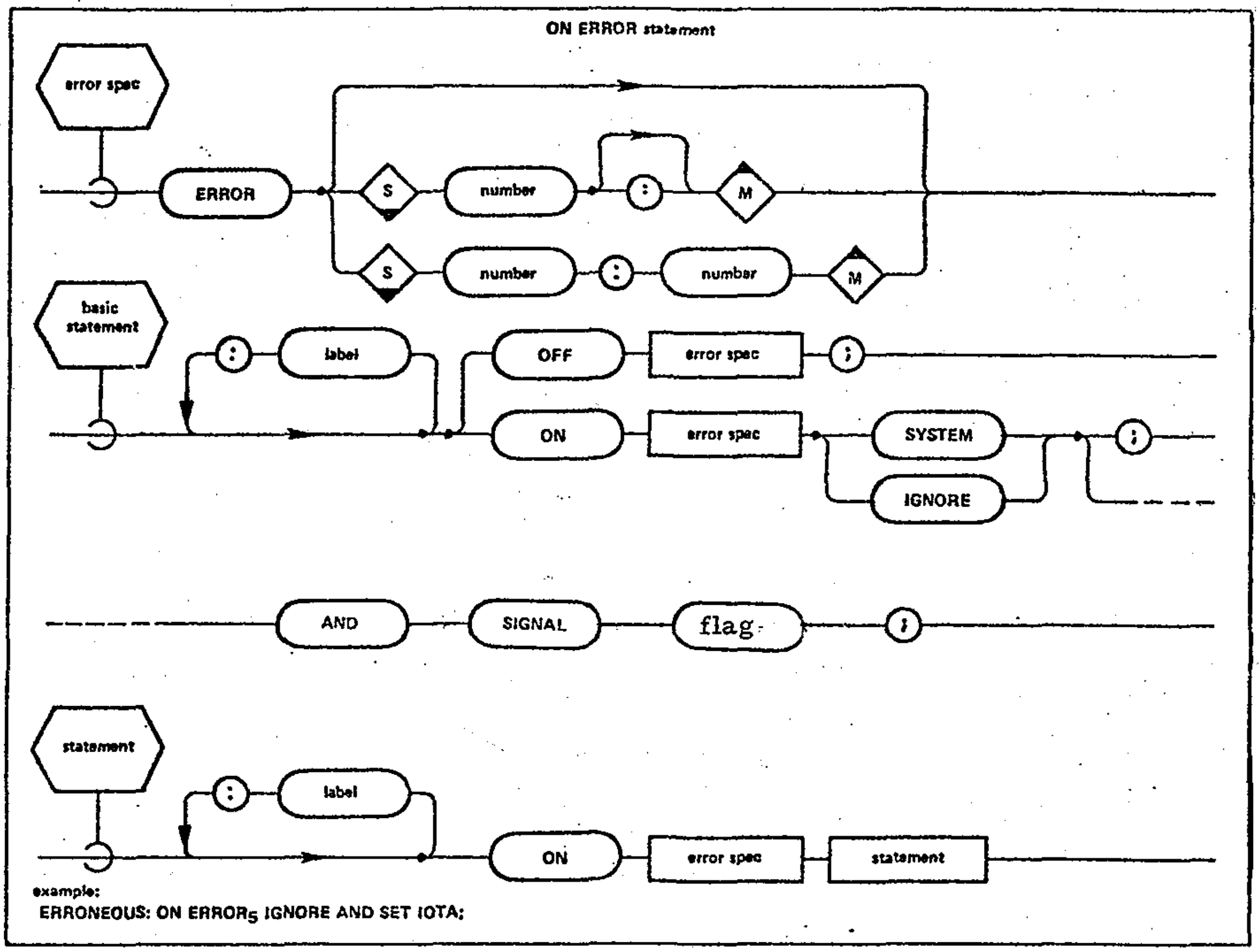
9.1 The ON ERROR Statement

The 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.

Error environment modification operates according to HAL/SM name-scope rules. If an ON ERROR with a given error specification is executed in a particular code block, then the modified recovery action remains in force until one of three things happens:

- o the modification is superseded by execution of a second ON ERROR with the same error specification,
- o 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, or
- o the modification is automatically removed by exit from the code block.

Syntax



ORIGINAL PAGE IS
OF POOR QUALITY

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, 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, or a selected error group, or a selected error code.

   o The form of <error spec> without subscript is used to specify all error groups.

   o 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.

   o 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 same 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 not possess statement labels. 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 clause. The purpose is to cause a specified program flag to be signaled on the occurrence of a specified error. Its semantic rules are the same as those described for the corresponding SIGNAL statement in Section 8.11.

9. The forms ON ERROR ... <statement> and ON ERROR ... IGNORE may not be allowed for certain errors.

<u>Precedence Rule</u>

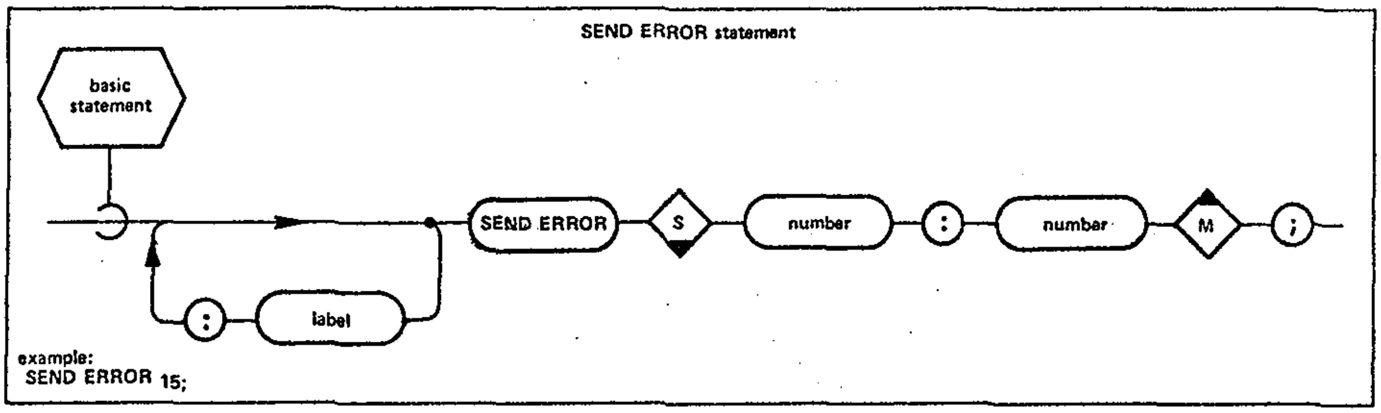
In a code block the action specified by an ON ERROR is only superseded by another if the two <error spec> s are of identical form. Similarly an OFF ERROR nullifies the effect of a previous ON ERROR only if the two <error spec> s are of identical form. However, different forms of <error spec> may involve the same error group or error code. It is logically possible for up to three ON ERROR's, each with a different form of <error spec> as described in Rule 2 above, to be active simultaneously and involve the same error code. 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> : <number> }	2
selected error code	<number> : <number>	3
		FIRST

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



Semantic Rules

1. A <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. INPUT/OUTPUT STATEMENTS

The HAL/SM language provides for two types of I/O: Standard Peripheral I/O Support (SPIOS) and Real Time I/O Support (RTIOS). Within SPIOS there are two forms: sequential I/O with conversion to and from external character string representation; and random-access record-oriented I/O. Within RTIOS there are four forms: Standard External Reference Point I/O (ERP I/O); Control and Display Console I/O (C&D I/O); system operator communication; and output of data to the system log.

All HAL/SM SPIOS is directed to one of a number of input/output "channels." These channels are the means used to interface HAL/SM software with standard peripheral devices in the run time environment. In the implementation each channel is assigned a unique unsigned integer identification number.

Most HAL/SM RTIOS is directed to one or more external reference points (ERP's). Statements which reference ERP's provide the capabilities necessary for controlling and using real time devices attached to the data bus of the Data Management Subsystem (DMS).

The Control and Display Console I/O statements are directed toward the Multifunction Display System (MDS) which is a specialized group of ERP's. These statements may be used for transmission of display information and MDS programs to the MDS and for receiving information from the MDS.

System Operator Communication I/O statements and System Log Output statements are directed toward devices dedicated for their purpose and are not directed toward ERP's.

10.1 SPIOS Sequential I/O Statements

All sequential I/O in HAL/SM is to or from character-oriented files. HAL/SM 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.

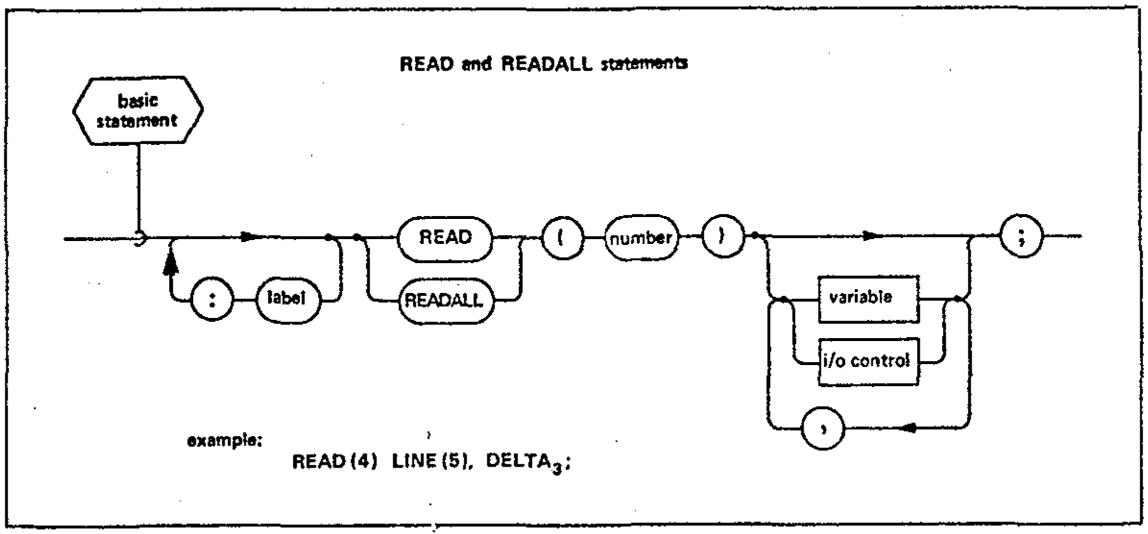
HAL/SM pictures the physical device as moving across the file a read or write "device mechanism" which actually performs the data transfer. The device mechanism has at every instant a definite column and line position on the file. The action of transmitting one character to or from the file is followed 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/SM 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/SM by employing either a READ or a READALL statement. The choice depends upon the format of the character input and the conversions (if any) which are to be performed. A READ statement is used wherever data in a standard external format is to be input; the READALL is used wherever arbitrary character string images are to be input without conversion.

Syntax



General Semantic Rules

1. A <number> is any legal I/O channel number.

2. An <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> before the first <variable>, 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 <i/o control> before the first <variable> overrides the automatic line advancement. A TAB or COLUMN <i/o control> overrides the automatic column positioning.

4. An unexpected end of file reached during the reading of data from the input file causes a run time error.

Semantic Rules: READALL Version

1. A <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, also 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.

Semantic Rules: READ Version

1. A <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. 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/SM data value. Fields may not cross line boundaries

except when reading character strings.

4. A semicolon field separator encountered during a normal sequential scan to fill a variable element terminates the READ statement as follows:

   o The current variable element is left unchanged.

   o All remaining< variable> s in the statement are unchanged.

   o All remaining control functions in the statement are ignored.

   The< i/o control> functions can force the device mechanism over the semicolon without causing early termination.

5. A null field is transmitted whenever a comma or a semicolon is detected when data is expected. This occurs when a comma or semi-colon is:

   o preceded by a comma or semicolon, or

   o preceded by one or more blanks following the last comma or semicolon.

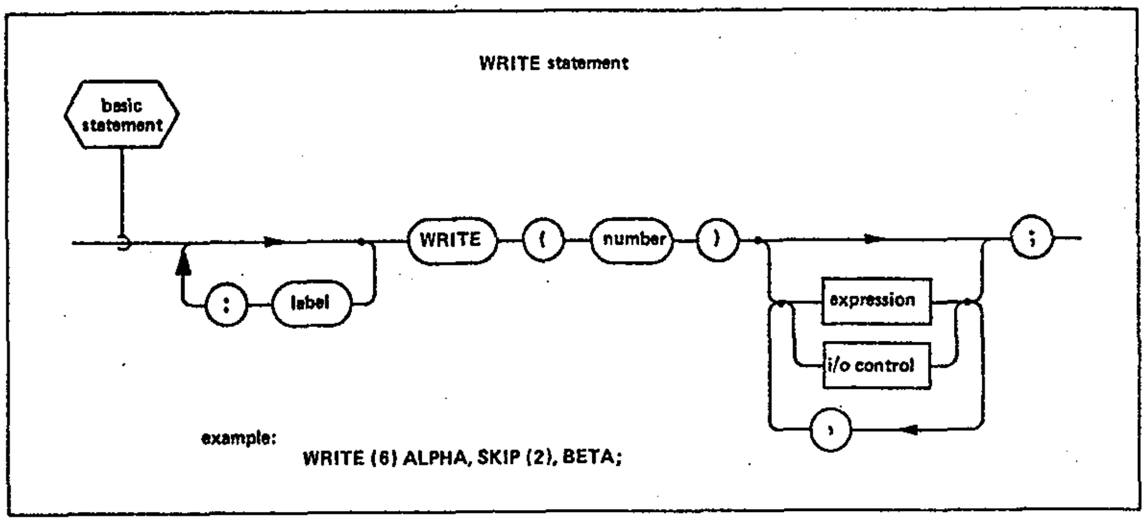
   A null field causes the corresponding variable element to remain unchanged following transmission.

6. Fields are assumed to be in a standard external format matching the type of each corresponding type of variable element. A mismatch between standard external format and element type causes a run time error.

10.1.2 The WRITE Statement

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

Syntax



Semantic Rules

1. A <number> is any legal I/O channel number.

2. An <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> before the first <expression>, 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 element in turn is converted to its standard external format before being transmitted to the output file.

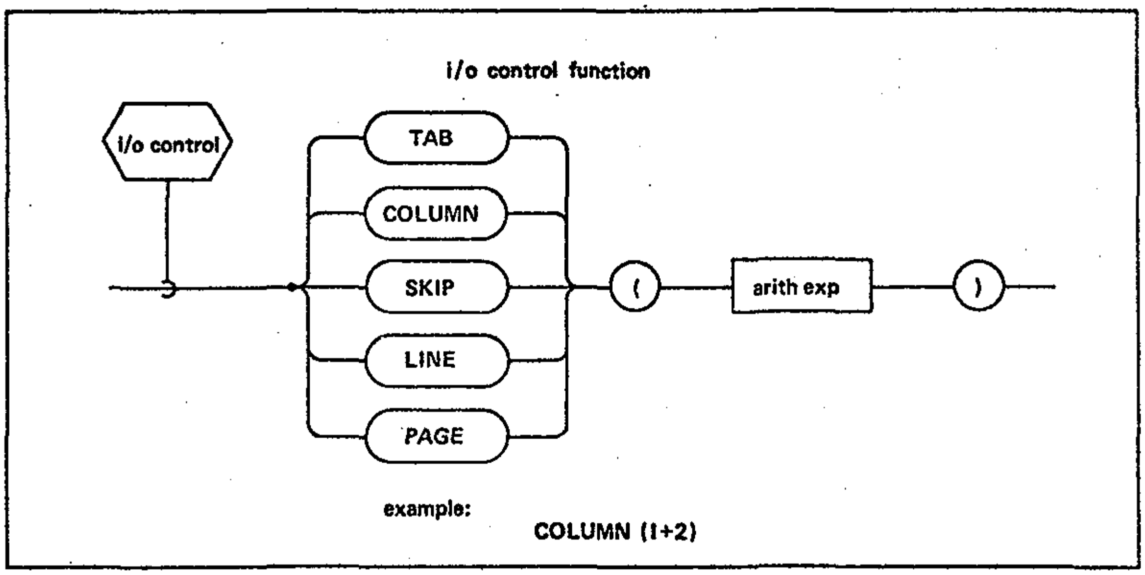
7. Between the transmission of two consecutive elements, the device mechanism is moved to the right by an implementation dependent number of columns. If a TAB or COLUMN <i/o control> separates two consecutive <expression> s then this overrides the automatic movement between transmission of the last element of the first <expression> and the first element of the second <expression> .

8. 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



Semantic Rules

1. An <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). 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 run time error.

3. COLUMN (K) specifies absolute movement of the device mechanism to colum 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 run time errors.

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.

   o 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.

   o 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 run time errors. Depending on the implementation, values of K causing backward movement may be illegal.

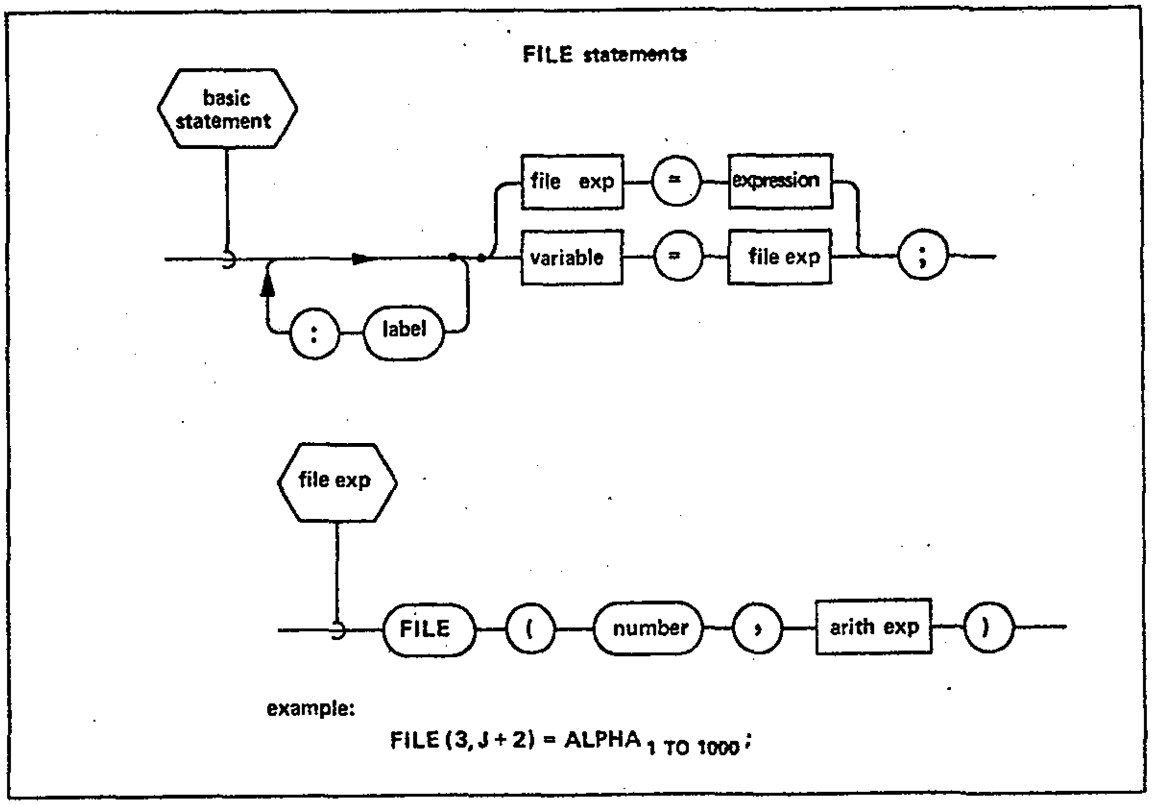
6. PAGE (K) is only applicable to paged files and specifies page movement relative to the current page. If K is positive the movement is forward, toward the end of file. Depending upon the implementation, negative page values may or may not be legal. The line value relative to the beginning of the page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

C.3

10.2 SPIOS 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



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. A <file exp> specifies the random access I/O channel and record address to be referenced. A <number> is any legal random access channel number. An <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 run time 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:

   - o No component subscripting for bit and character types.
   - o If component subscripting is present, <variable> must be subscripted so as to yield a single (unarrayed) element of the <variable> .
   - o If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.
   - o 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.
   - o If the <variable> 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.

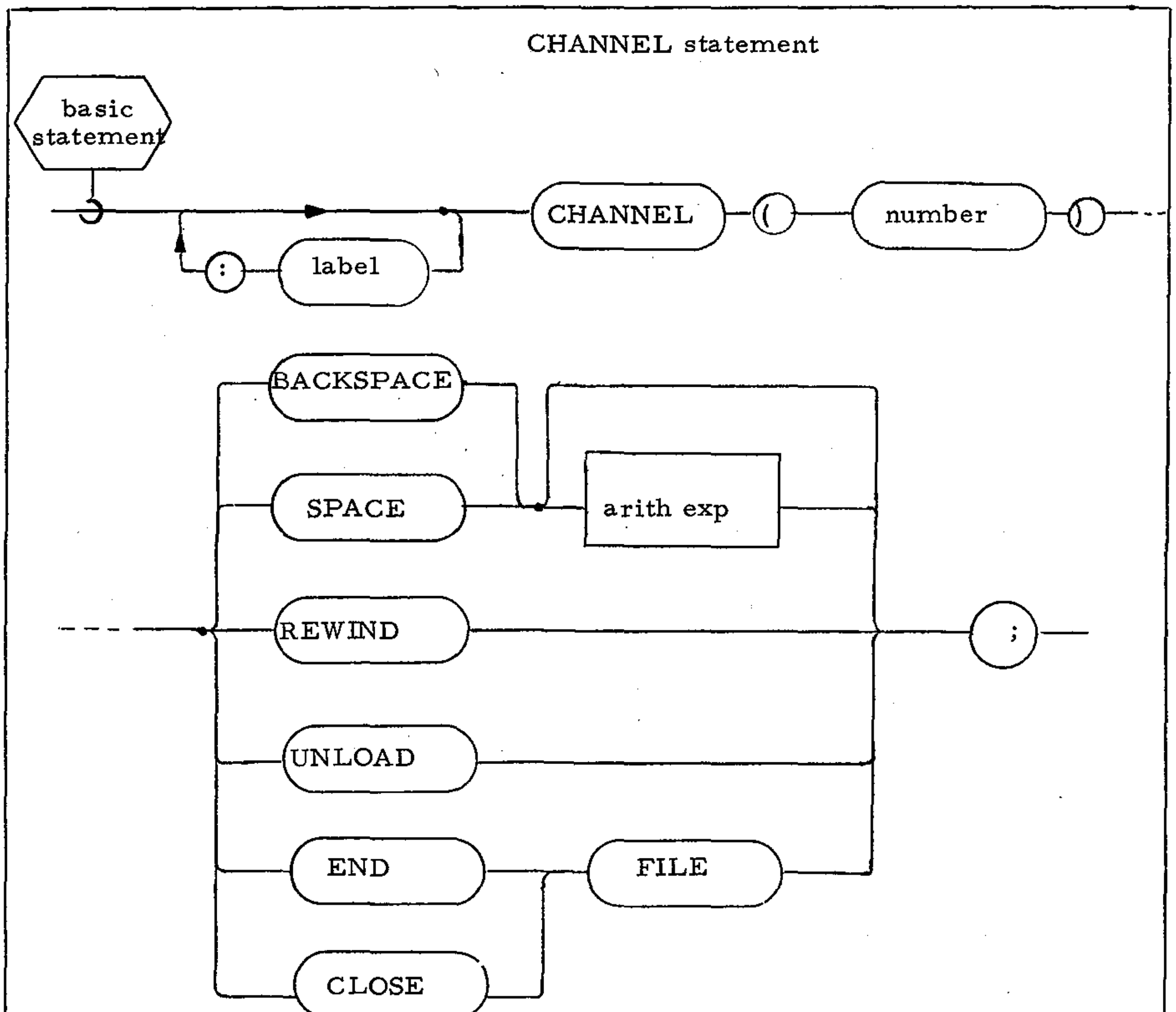
5. In the output FILE statement, there are no semantic restrictions on < expression> .

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.

10.3 SPIOS Channel Control and the Channel Statement

Certain I/O operations are common to all channels independent of whether sequential or random access I/O is being used. These operations may be performed using the Channel Statement.

Syntax



example:

```
        CHANNEL (3) REWIND;
```

Semantic Rules

1. An <arith exp> specifies the number of files to be forward spaced or backspaced. If it is omitted, a default value of one is used. Its value must be greater than zero, otherwise a run time error is generated.

2. UNLOAD may only be used on tape files. If used for any other device type, a run time error is generated.

3. REWIND and BACKSPACE may only be used on unpaged channels.

4. The CLOSE FILE operation yields an end-of-file and deallocation of the file.

10.4 RTIOS ERP I/O Statements

ERP I/O statements are used to interface with the following types of ERP's:

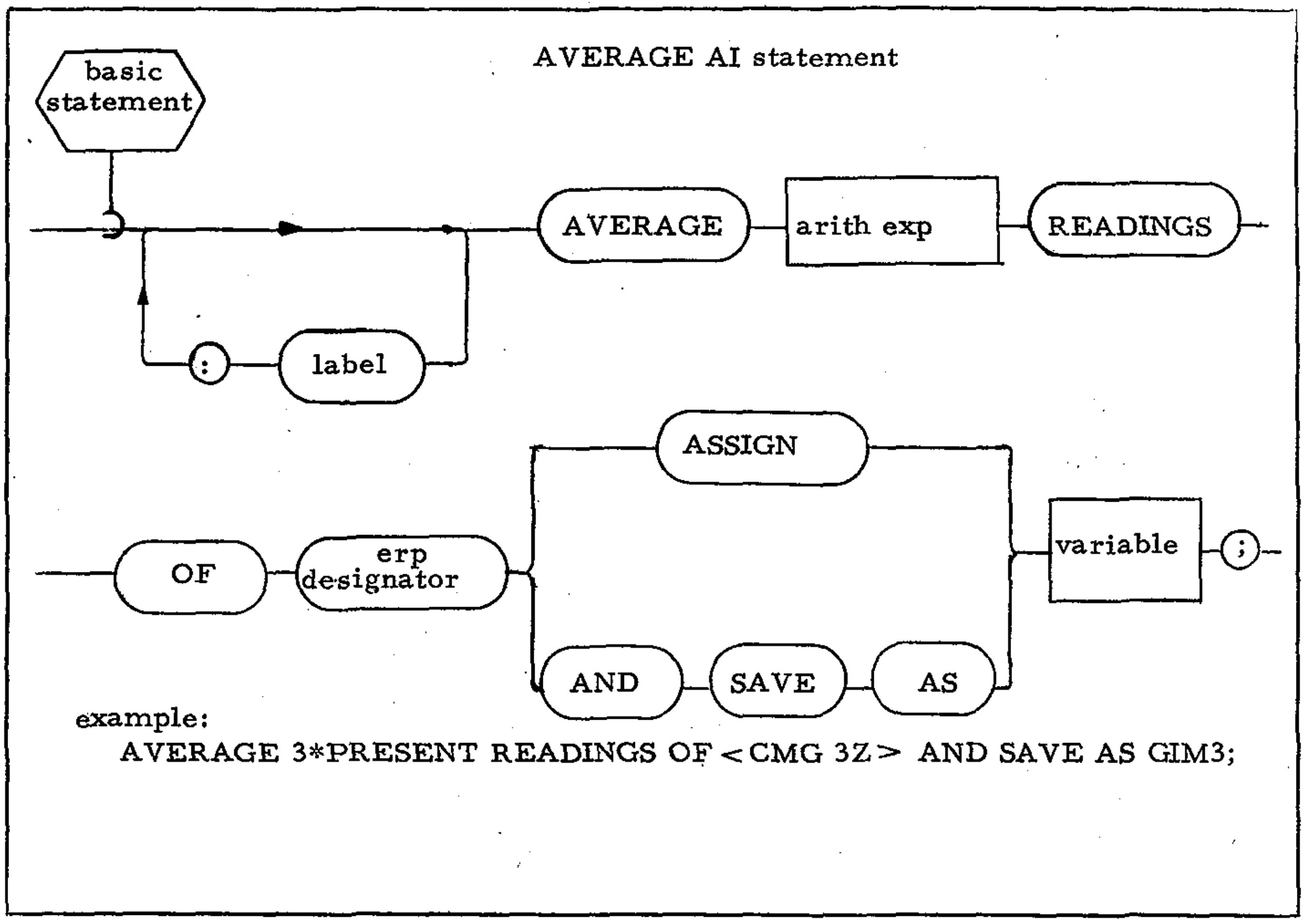
- o Discrete Input (DI)
- o Discrete Output (DO)
- o Analog Input (AI)
- o Analog Output (AO)
- o Record Input (RI)
- o Record Output (RO)

The ERP I/O statements have several forms which are determined by the type of ERP to which they are directed. All ERP I/O statements are specified using the ERP names defined in the Measurement and Control Definition (M&CD). The HAL/SM system uses the specified ERP name and the M&CD to validate the type of ERP I/O statement being used and the data passed to or from the specified ERP.

10.4.1 The AVERAGE AI Statement

The AVERAGE AI statement provides a method of averaging several analog (AI) measurements from the same ERP. This statement is used whenever a simple mathematical average - the sum of the measurements divided by the number of times read - is required.

Syntax



Semantic Rules

1. An <arith exp> is an unarranged scalar or integer arithmetic expression specifying the number of readings to be used. The value is treated as an integer; scalar values are rounded to the nearest integer prior to use.

2. A <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:

   o No component subscripting for bit and character types.

   o If component subscripting is present, <variable> must be subscripted so as to yield a single element of the <variable> .

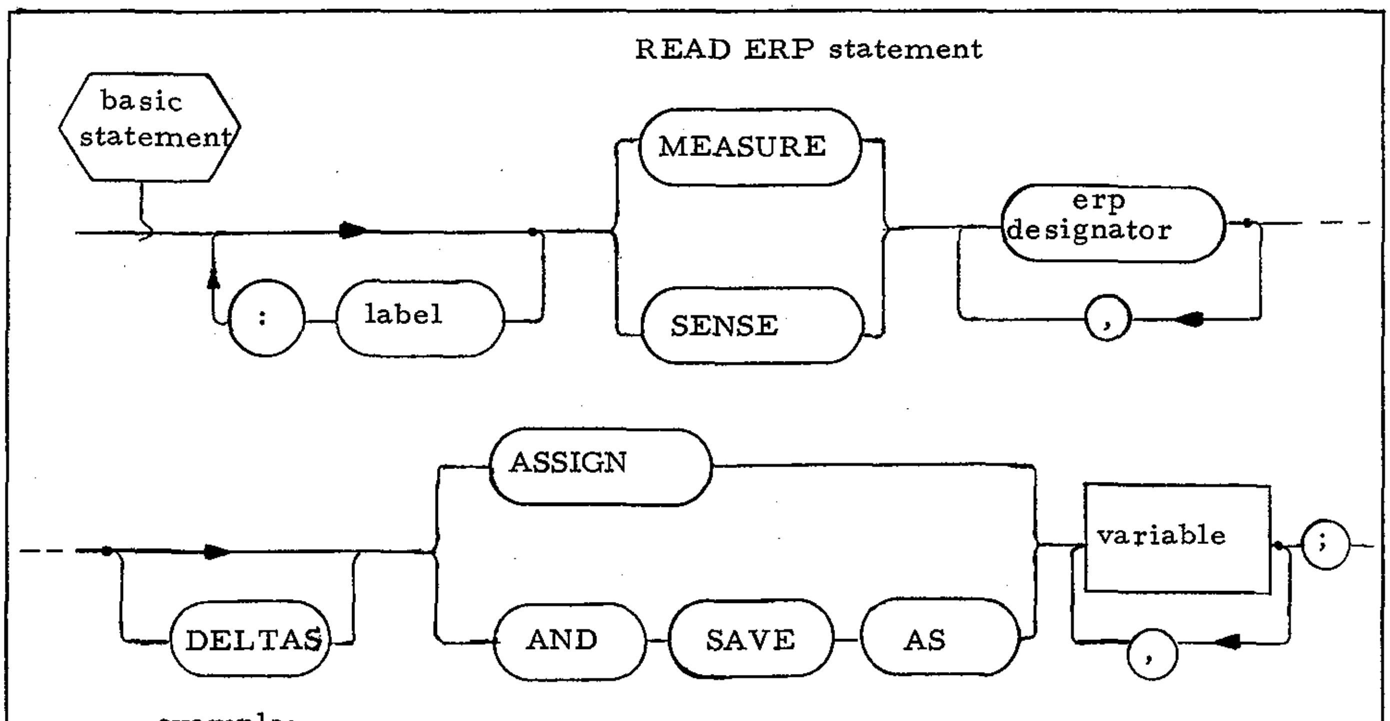
   o If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

- 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 if it is a major structure) and if the structure possesses multiple copies, then subscripting must reduce the copies to one.

10.4.2 The READ ERP Statement

The READ ERP statement acquires ERP data and stores it as internal program data.

Syntax



example:
MEASURE <THRUST> AND SAVE AS T15;

Semantic Rules

1. If more than one <erp designator> is specified, all ERP's designated must be of the same ERP type.

2. The term "DELTAS" only applies to ERP's of type AI.

3. Two <variable> s or a single <variable> array of two elements must be specified for AI DELTAS.

4. If more than one <variable> is specified, there must be exactly the same number of <variable> s and <erp designator> s.

5. A <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.

   o No component subscripting for bit and character types.

   o If component subscripting is present, <variable> must be subscripted so as to yield a single element of the <variable> .

   o If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

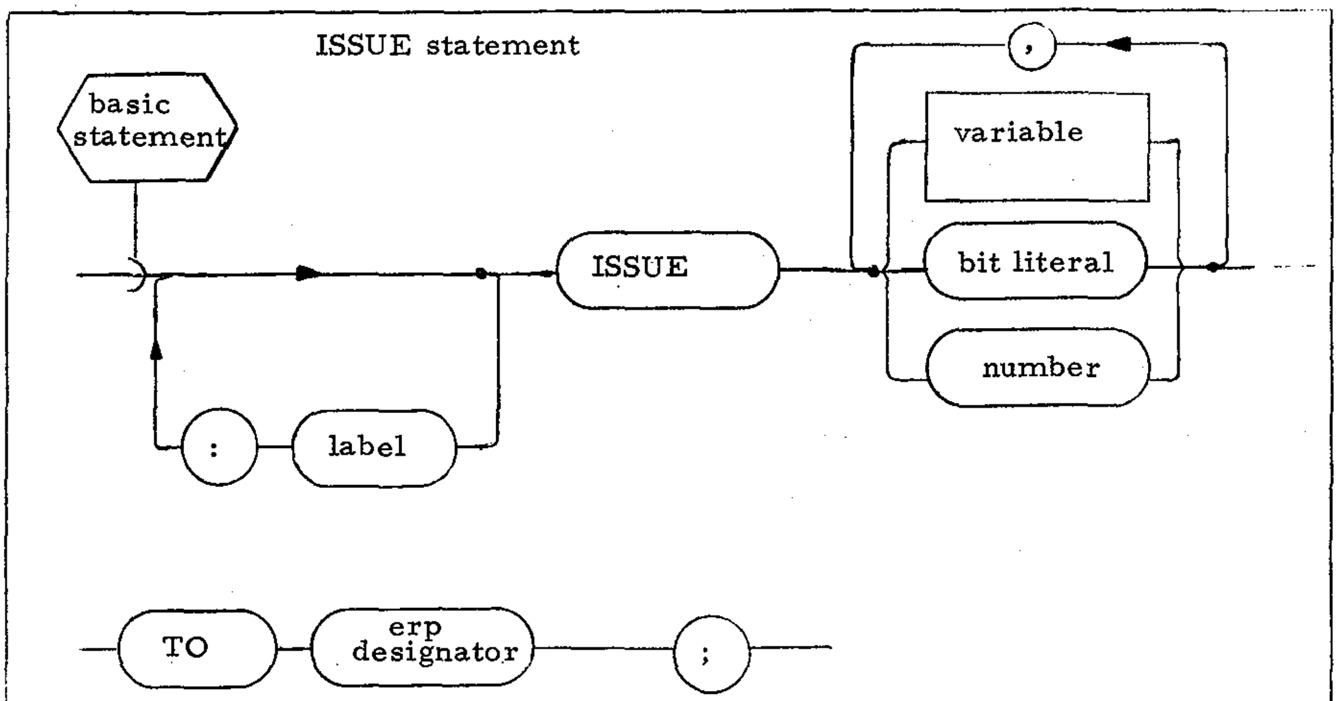
   o BIT type structure terminals which have the DENSE attribute may not be used, due to packing implication. However, an entire structure with the DENSE attribute may be used.

   o If the <variable> 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 subscripting must reduce the copies to one.

10.4.3 The ISSUE Statement

The ISSUE statement allows for the transmission of a Record Out (RO). The record is limited to an implementation dependent size.

Syntax



example:

ISSUE BILL TO <RECORD 1> ;

Semantic Rules

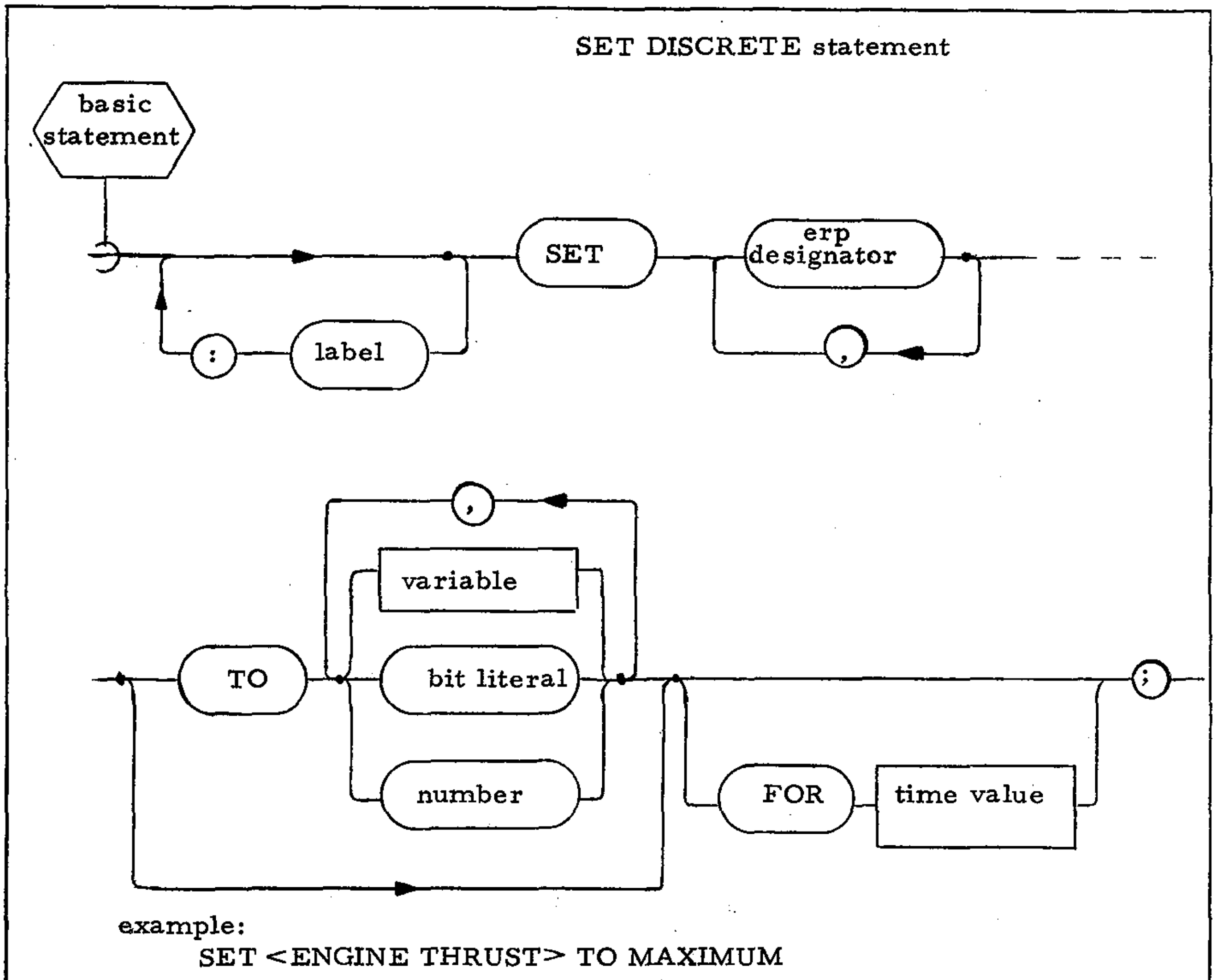
1. If multiple values (i.e., variables or literals) are specified, no <variable> may be arrayed. Also the multiple copies must have the same type and attributes with the exception that character strings may be variable length.

2. Multiple copies of <bit literal> must be of the same length.

10.4.4 The SET DISCRETE Statement

The SET DISCRETE statement allows for setting or pulsing a Discrete Output (DO).

Syntax



Semantic Rules

1. If the DO is to be pulsed, only one < erp designator > may be specified.

2. If multiple values (i.e., variables or literals) are specified, no < variable > may be arrayed, also, there must be a one to one correspondence between each < erp designator > and < variable > specified.

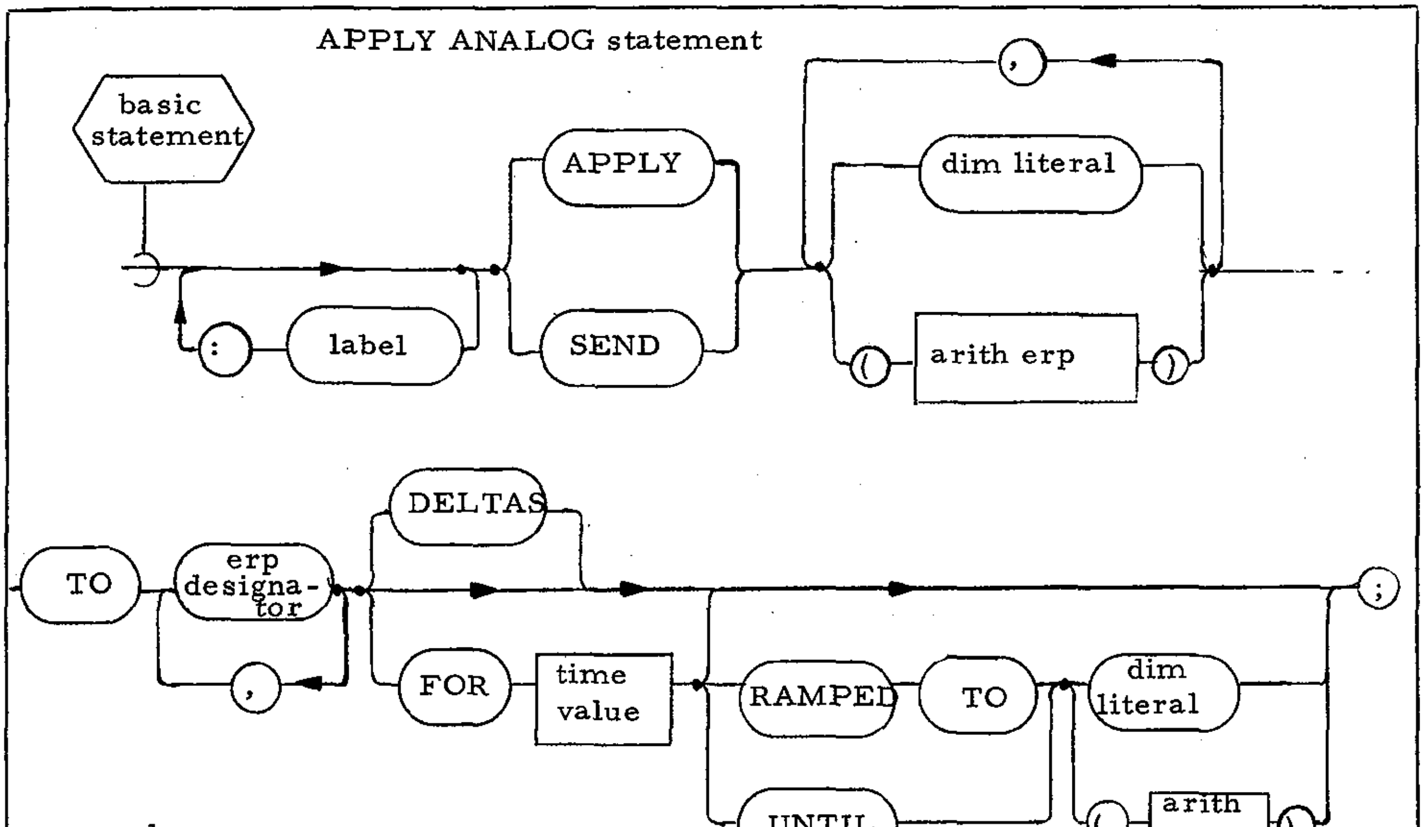
3. If an arrayed < variable > is specified, there must be an array element for each < erp designator > specified.

4. When the TO phrase is omitted the < erp designator > is said to be self-defining and its value is retrieved from the on-line M&CD.

10.4.5 The APPLY ANALOG Statement

The APPLY ANALOG statement provides the means for producing an analog output (AO) and/or an AI delta. The value of the AO may be set once, pulsed, or ramped.

Syntax



example:
APPLY IOV TO<RESISTOR1> FOR 10 SECS;

Semantic Rules

1. If multiple values (< dim literal> and/or < arith exp>) are specified, a one to one correspondence must exist with an < erp designator> .

2. Exactly two values must be specified when statement is SEND (or APPLY) AI DELTAS.

3. Only one < erp designator> may be ramped or pulsed. In either of these cases, only one value may follow SEND or APPLY.

4. The value following RAMPED TO or UNTIL specifies the maximum (or minimum if first value negative) to which the <erp designator> is to be ramped.

10.5 *RTIOS C&D Console I/O Statements*

The C&D Console I/O statements allow communication with the MDS by sending and receiving information to and from a portion of the MDS memory known as the display buffer. This buffer is subdivided into units identified as CRT (Cathode Ray Tube) pages. Each I/O statement is directed toward one CRT page in the display buffer.

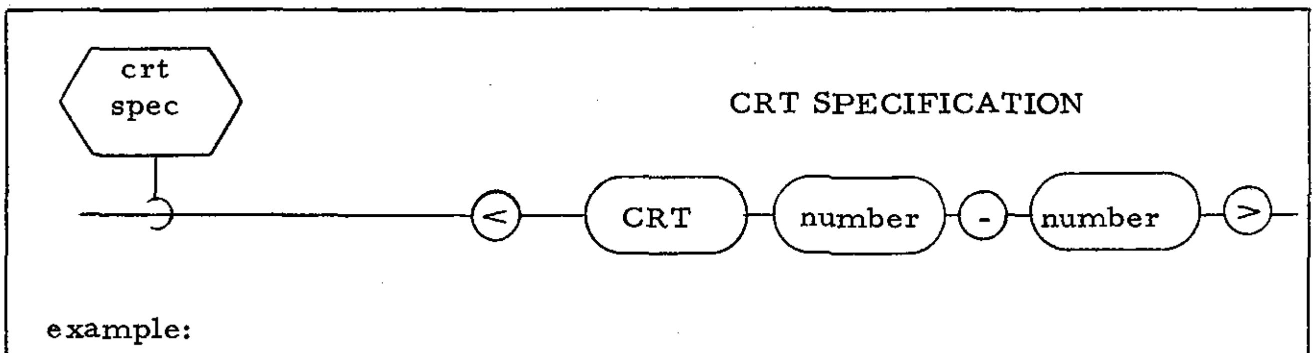
The Control and Display Console I/O statements provide means to:

- o request data from a CRT page,
- o control CRT pages,
- o transmit background and add on display information to CRT pages,
- o transmit and execute MDS programs on CRT pages,
- o update background display information, and
- o update MDS program data.

10.5.1 The CRT SPECIFICATION

In order to expedite the definition of a CRT page of the display buffer, the syntactical term < crt spec > is introduced.

Syntax



example:
< CRT 2-2 >

Semantic Rules

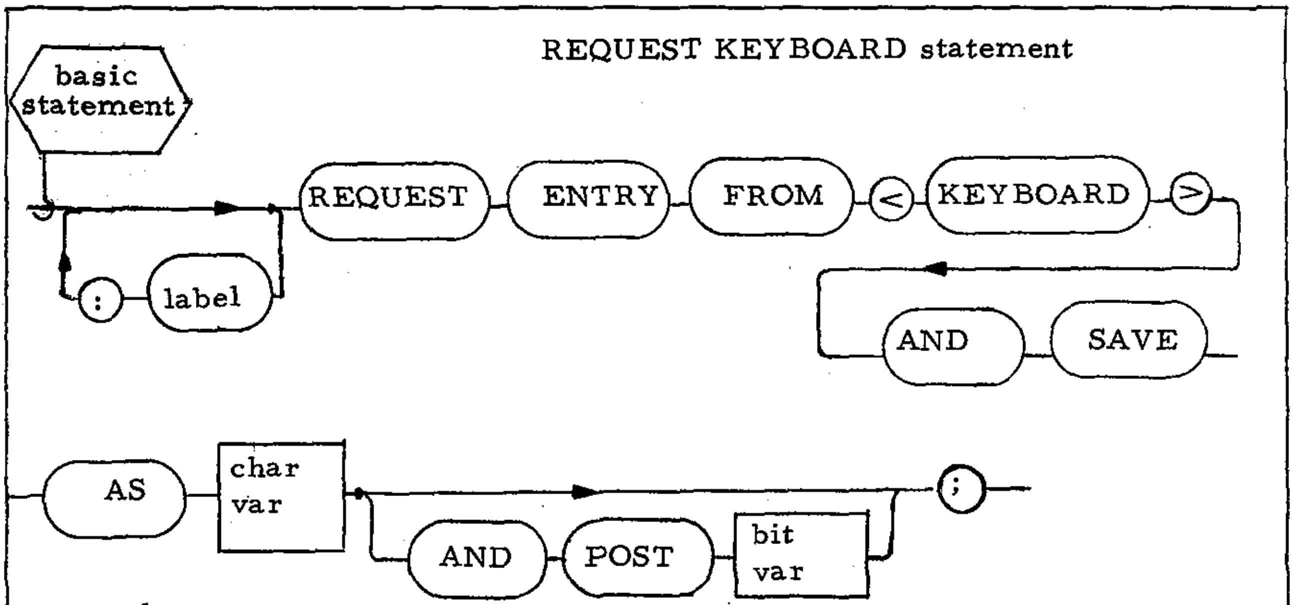
1. The < number > following CRT identifies the C&D console being accessed; whereas, the < number > following - identifies the page being accessed.

2. A < number > is a positive integer in each case.

10.5.2 The REQUEST KEYBOARD Statement

The REQUEST KEYBOARD statement allows data to be requested from a particular CRT page. The statement may be used in conjunction with the DISPLAY DATA statement to carry out a tutorial from a C&D console.

Syntax



example:
REQUEST ENTRY FROM <KEYBOARD> AND SAVE AS YAWX1;

Semantic Rules

1. The CRT page from which data is returned is identified as the master page. There is only one master page per job.

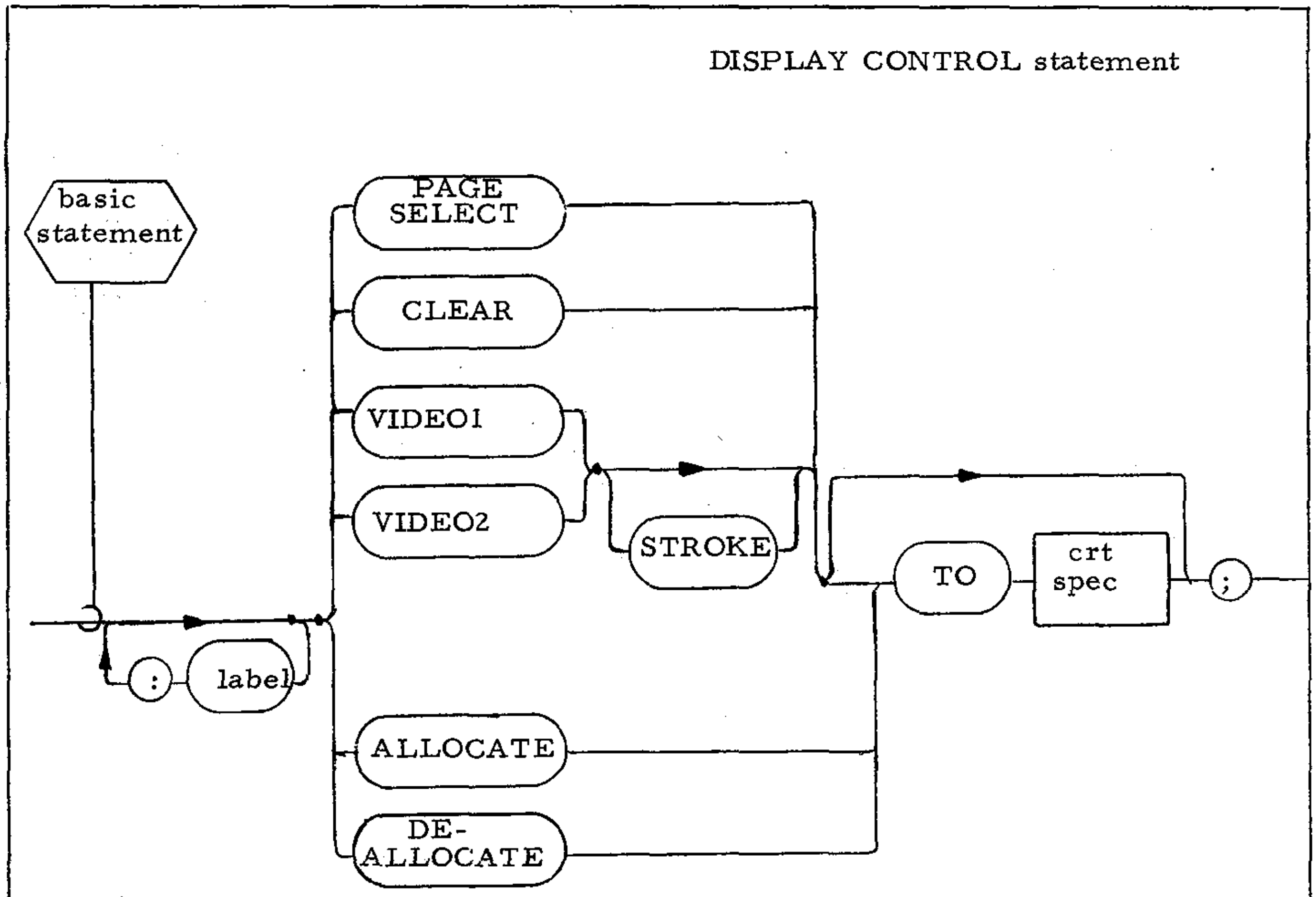
2. If the AND POST <bit var> portion of the statement is omitted and no data is available for response to the request, the requesting task is placed in an automatic wait with no notification to the task.

3. The <bit var> in the AND POST clause must be Boolean (i.e., BIT (1)).

10.5.3 The DISPLAY CONTROL Statement

The DISPLAY CONTROL statement provides the interface to effect CRT page control. The statement allows requests for CRT page allocation/deallocation, CRT page clear, and page selection. If the requested page is not available, the request fails. Also a combination of the VIDEO/STROKE commands enables television reception and display data to reside simultaneously on a specified CRT.

Syntax



example:

DEALLOCATE <CRT 2-9> ;

Semantic Rules

1. If no <crt spec> is identified, the default value is the master CRT page which has been assigned by the C&D Subsystem to the particular job being executed.

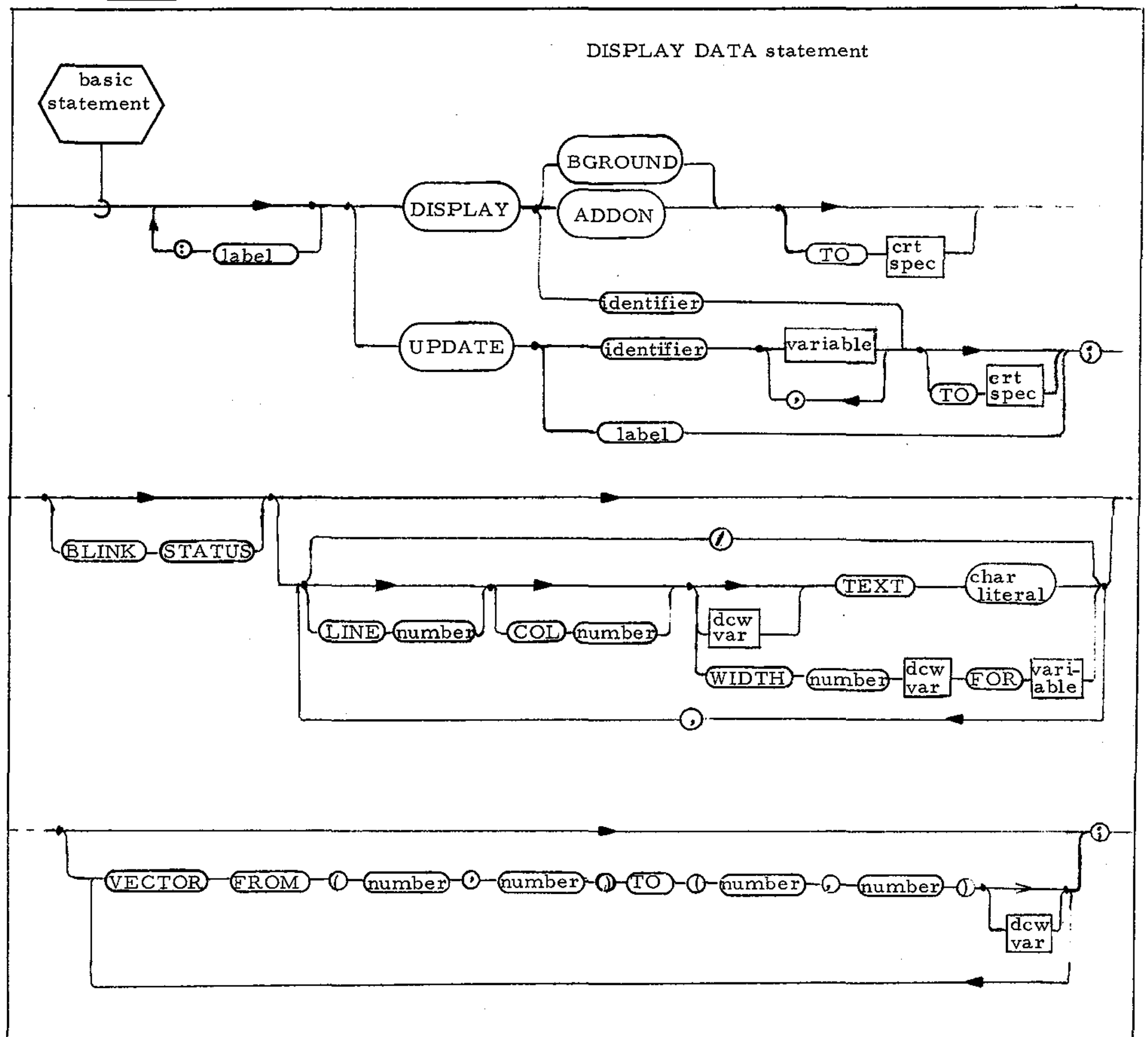
2. As noted on the syntax diagram, the master page may not be allocated or deallocated.

10.5.4 The DISPLAY DATA Statement

The DISPLAY DATA statement provides for the output of information to a CRT page. This includes:

- o Formatting and displaying background data comprised of text and/or vectors.
- o Providing an update option for variable data in background displays.
- o Providing an add on message capability with optional variable data.
- o Initiation of MDS display program execution.
- o Providing an update option to supply data parameters to MDS display programs.

Syntax



example: DISPLAY PGMTP1 TO <CRT2-1> ;

ORIGINAL PAGE IS
OF POOR QUALITY

Semantic Rules

1. The <identifier> specifies the MDS display program name. This name must comply to implementation dependent restrictions.

2. If TO <crt spec> is omitted, the master page is assumed.

3. The <label> following UPDATE must reference a previously defined DISPLAY DATA statement.

4. If BLINK STATUS is not specified, no blinking is assumed. However, this may be overridden by any <dcw var> which is included in the DISPLAY DATA statement.

5. Any <number> specified must be a positive integer.

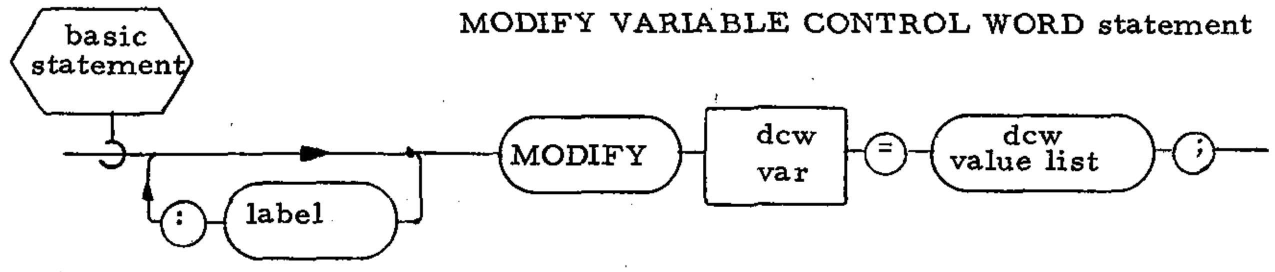
6. When a <dcw var> is used with TEXT or VECTOR, it represents a text control word for which the output format and variable data width are not applicable.

7. A "/" is an end-of-line indicator which may be used as a carriage return signal.

10.5.5 The MODIFY VARIABLE CONTROL WORD Statement

The MODIFY VARIABLE CONTROL WORD statement provides a method for real time modification to variable control words. The specifications which may be changed are the color, character size, intensity, output format, and blink status.

Syntax



example:
MODIFY VARCW6 = YELLOW, 10MM, BLINK OFF, 6;

Semantic Rules

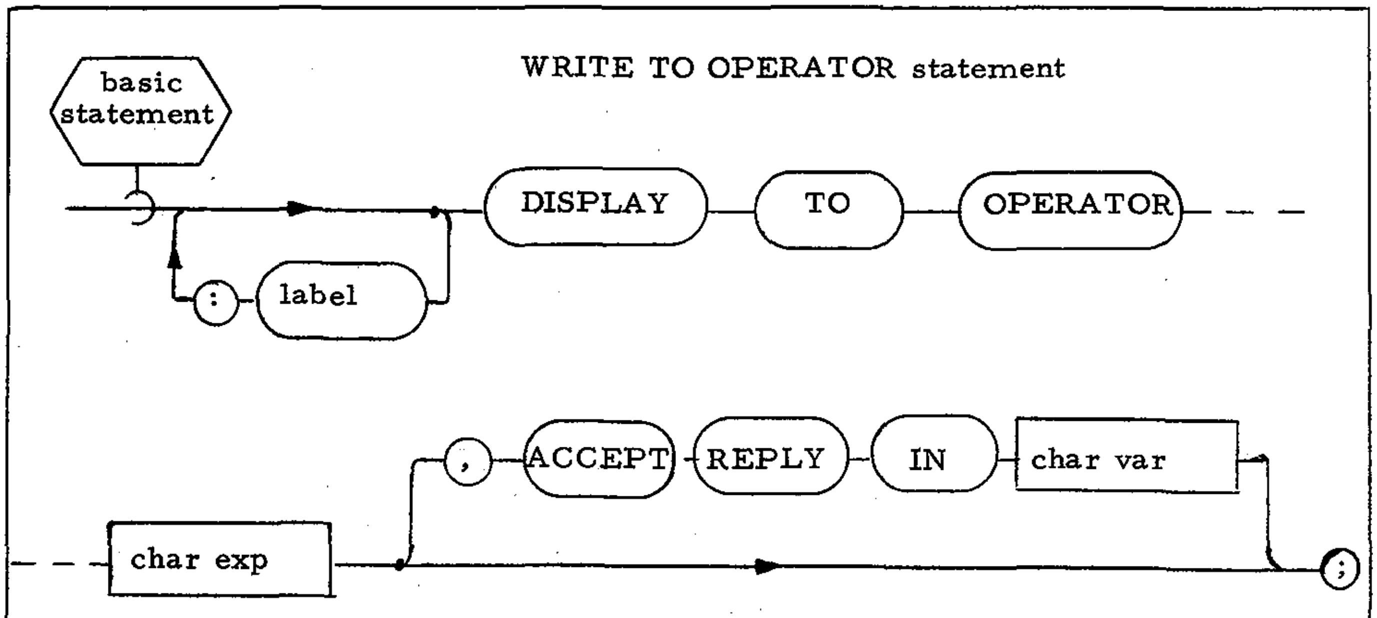
1. The < dcw var > must identify a variable control word.

2. Any options omitted in the <dcw value list> will default to the previous values of the <dcw var> .

10.6 RTIOS System Operator Communication I/O Statement

An I/O statement is provided to enable a task to communicate with the system operator. The statement is directed toward a system operator console.

Syntax



example:
DISPLAY TO OPERATOR 'TASK TERMINATED';

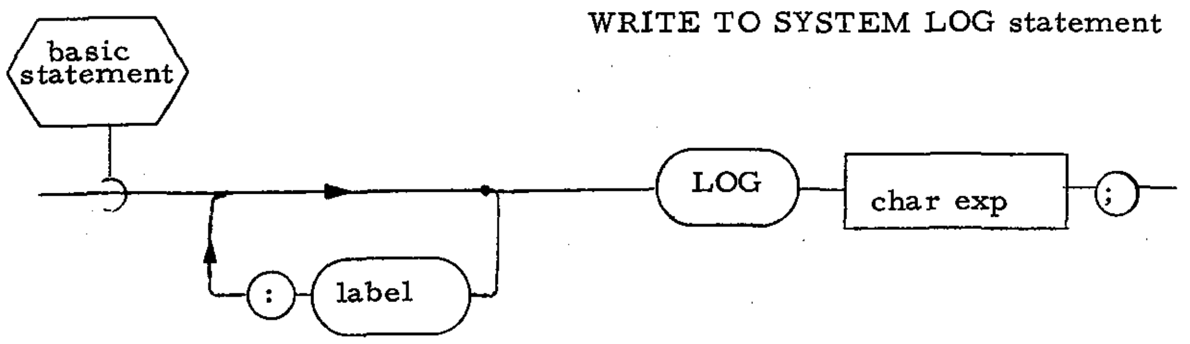
Semantic Rules

1. If the ACCEPT REPLY IN <char var> clause is specified, the task executing the statement is placed in a wait state until the system operator supplies a reply.

10.7 RTIOS Output to the System Log

Information may be entered into the system log by using the Output To System Log statement.

Syntax



example:

```
    LOG 'TASK #5 EXECUTING';
```

11. SYSTEMS LANGUAGE FEATURES[1]

11.1 Introduction

The systems language features of HAL/SM are described in this section. The features presented here are in three sections. A new program organization feature is provided by "Inline Function Blocks." A data-related feature of this systems language extension is the concept ot "TEMPORARY variables." The NAME Facility concerns a new concept in HAL/SM, the addition of NAME variables pointed 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.

11.2 Program Organization Features

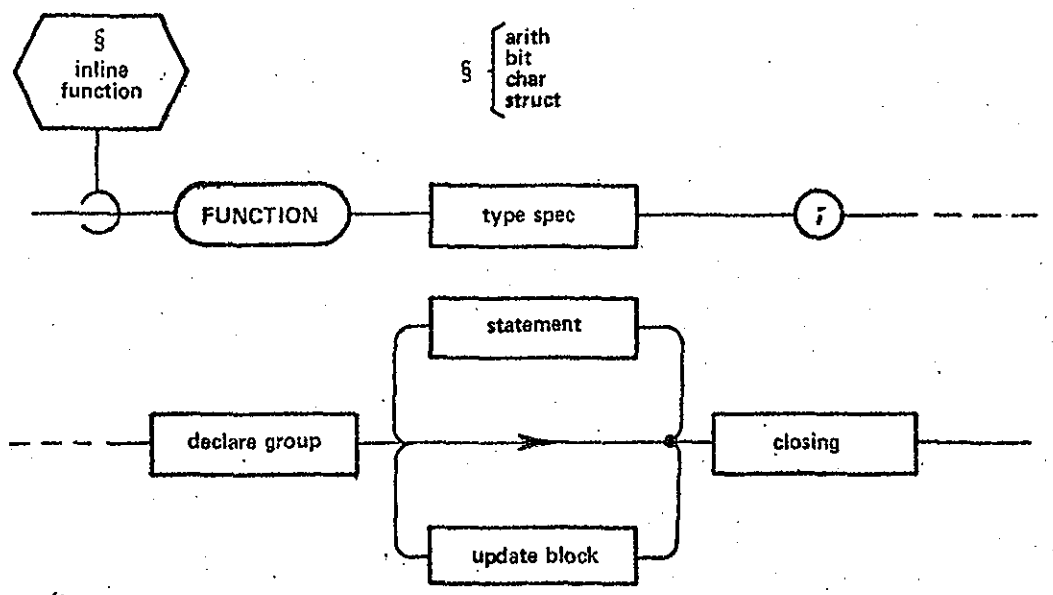
The addition of Inline Function Blocks to HAL/SM extends the informa tion presented in Section 3 concerning program organization. Inline functions are a modified kind of user function in which invocation is simultaneous with block definition.

11.2.1 Inline Function Blocks

The HAL/SM 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 parametric REPLACE statement described in Section 4.2. Its appearance is generally in the form of an operand of an expression.

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

Syntax



example:

```
IF X ¬= Y THEN R = FUNCTION VECTOR;
                   DECLARE A,B;
                   A = 3X + Y;
                   B = X/Y,
                   RETURN VECTOR(A,B,0);
                   CLOSE;
T = R*V;
  .
  .
  .
```

Semantic Rules

1. The syntactical form is actually equivalent to that of a function block except that:

   o The <§ inline function> has no label,

   o The <§ inline function> has no parameters, and

   o 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 restrictions listed below.

3. A < g inline function> may not contain the following syntactical forms:

   o All forms of I/O statements,

   o All forms of reference to user-defined PROCEDURE and FUNCTION blocks, and

   o Real Time and Error Recovery and Control statements.

4. A < g inline function> may not contain any form of nested blocks. The following block forms are thus excluded:

   o < function block> definitions, and

   o < procedure block> definitions,

   o <update block> , and

   o Further nested < g inline function> s.

5. In use, the following semantic restriction holds: < g inline function> s may not appear as operands of the subscript or exponent expressions.

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

   o <arith inline> - <type spec> specifies an inline function of an arithmetic data type: SCALAR, INTEGER, VECTOR or MATRIX.

   o < bit inline> - < type spec> specifies an inline function of a bit type: BOOLEAN or BIT.

   o < char inline> - < type spec> specifies an inline function of the CHARACTER data type.

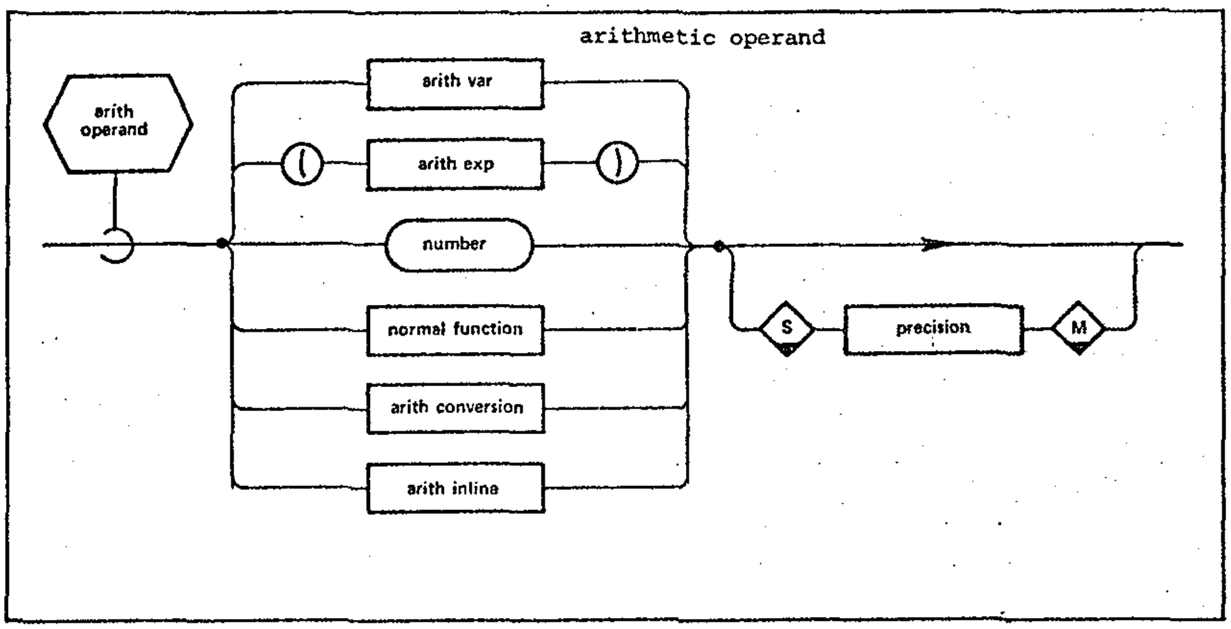
   o < struct inline> - < type spec> specifies an inline function with a structure type specification.

   The use of inline functions as operands of HAL/SM expressions is discussed in Section 11.2.2.

11.2.2 Operand Reference Invocations

Inline Function Blocks are always invoked at the point of their definition as operands of < expression > s. 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 Operand

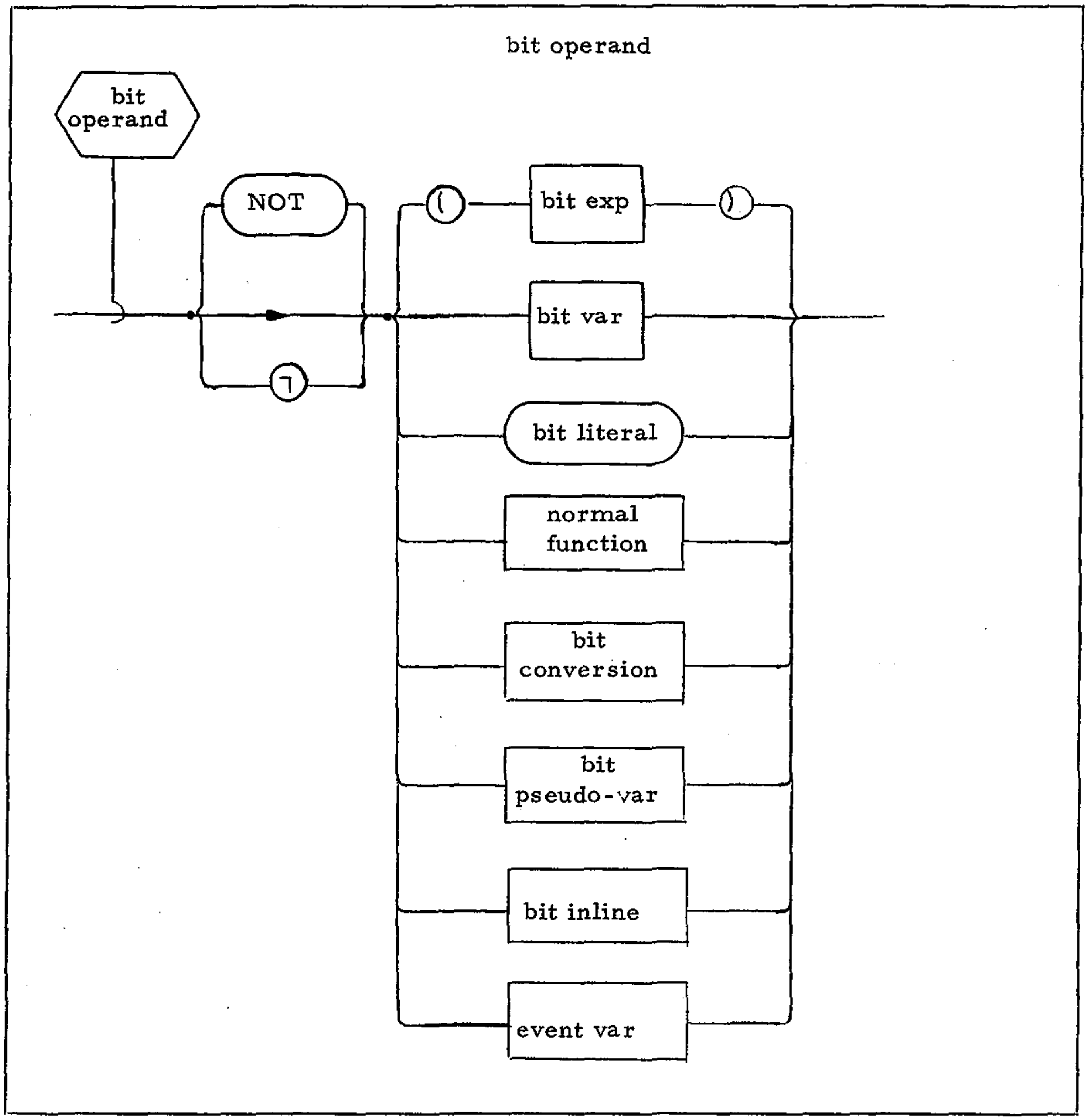


Semantic Rules

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. An <arith inline> is an inline function block which has an arithmetic < type spec> in its header statement.

Syntax of Bit Operand

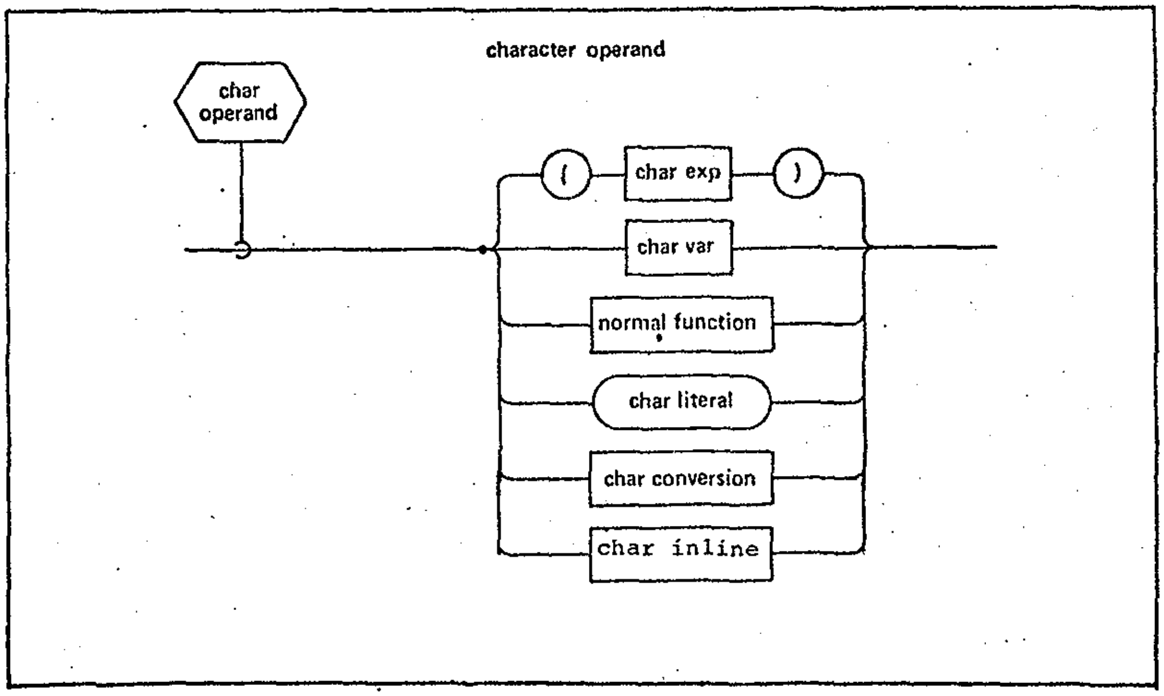


Semantic Rules

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. A <bit inline> is an inline function block which has a bit string (BOOLEAN or BIT) <type spec> in its header statement.

Syntax of Character Operand

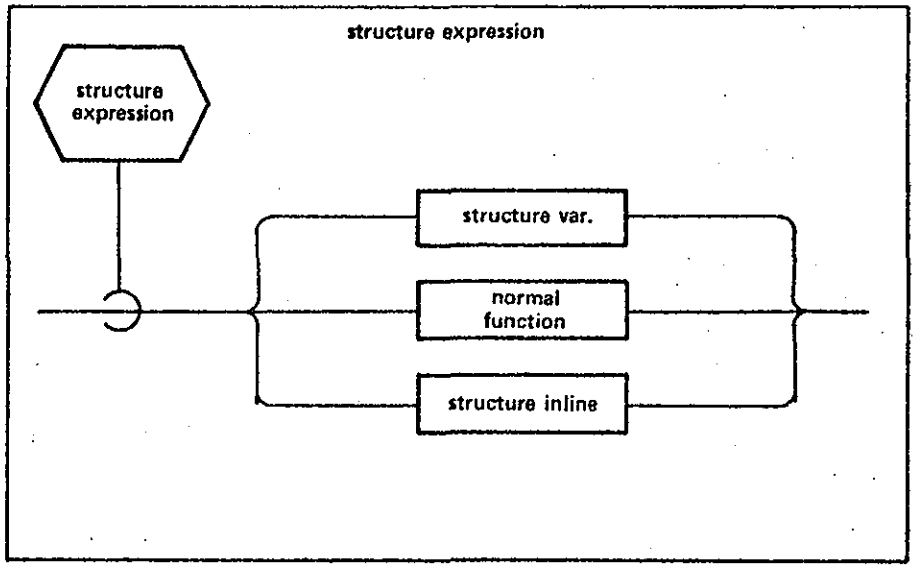


Semantic Rules

1. This 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. A <char inline> is an inline function block which has a CHARACTER <type spec> in its header statement.

Syntax of Structure Expression



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. A <struct inline> is an inline function block which has a structure < type spec> in its header statement.

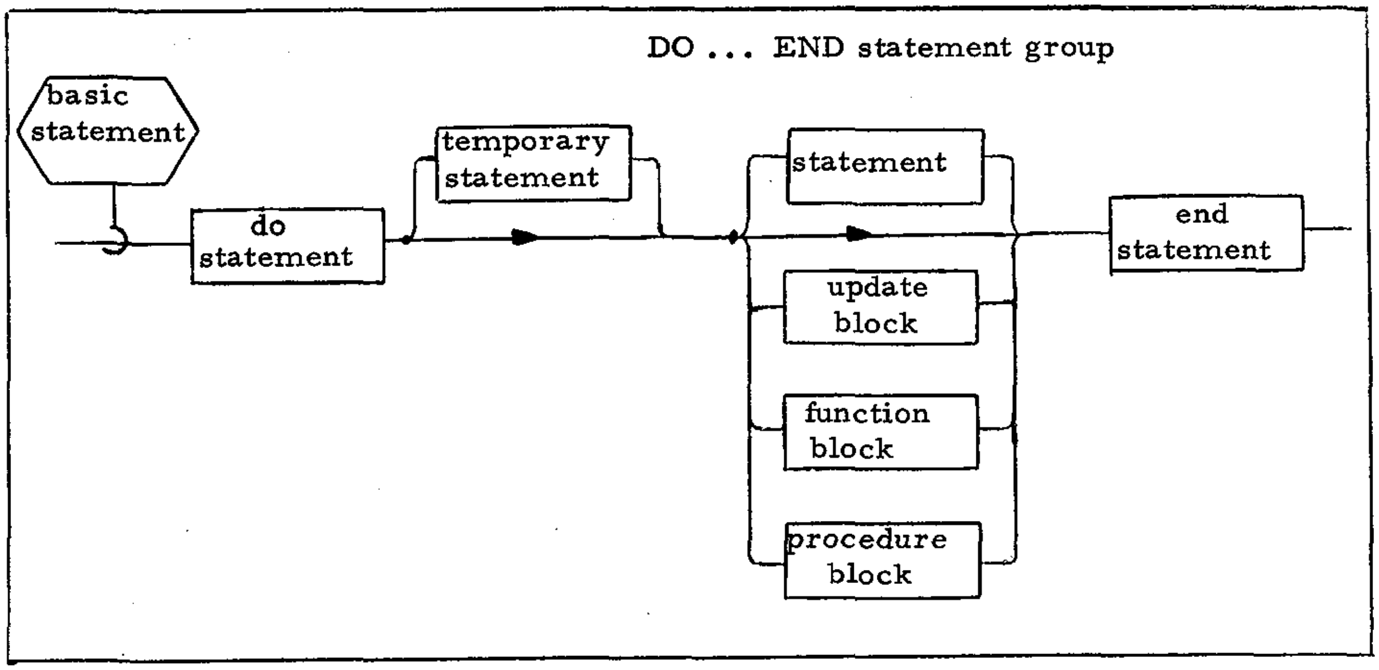
11.3 Temporary Variables

The extension of HAL/SM 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, scratchpad 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 scratchpad 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.

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

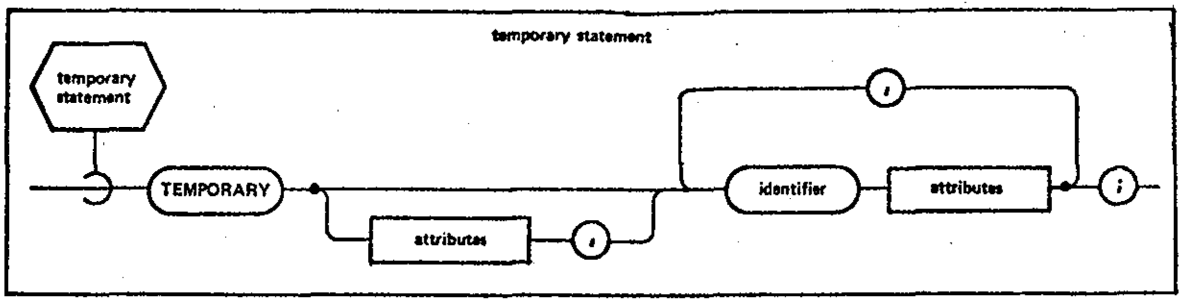


Semantic Rule

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



Semantic Rules

1. In the <temporary statement>, <attributes> may define the <identifiers> to be of any data type except EVENT, DCW, or FLAG.

2. The <attributes> may only specify type, precision and arrayness.

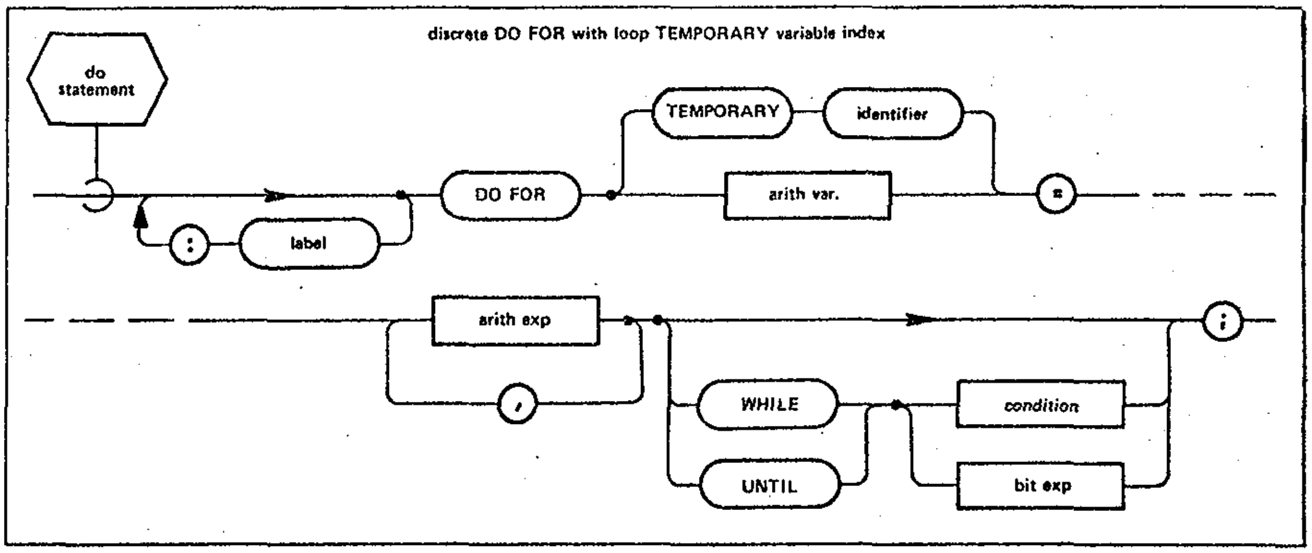
3. No minor attribute is legal.

4. The name of <identifier> may not duplicate the name of another <identifier> in the same name-scope (procedure, function, or other block) or of another temporary in the same DO ... END group.

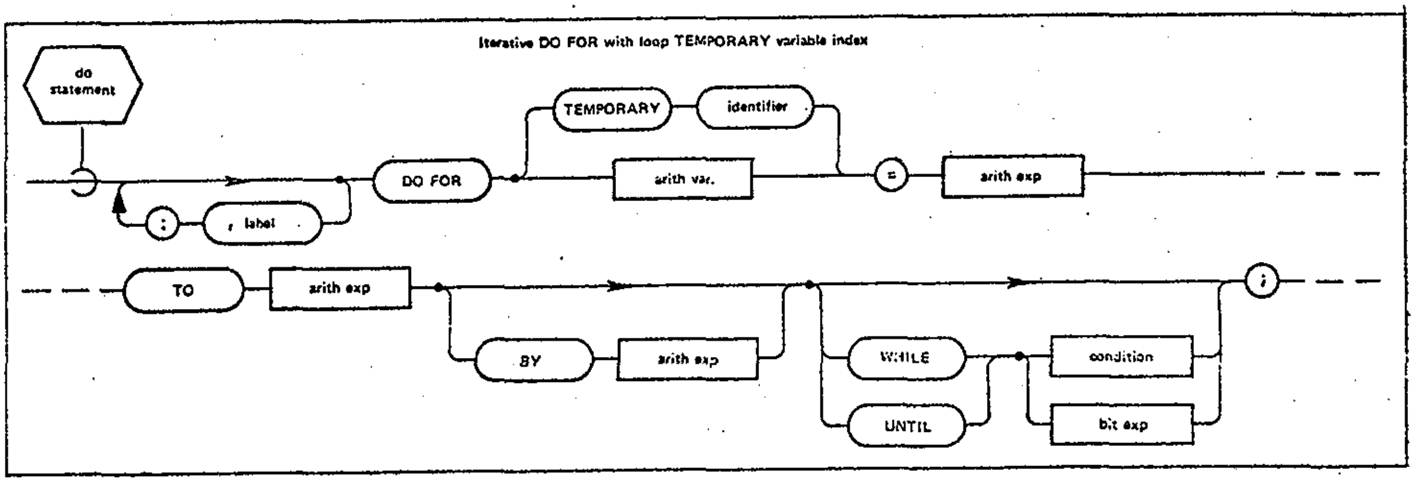
11.3.2 Loop TEMPORARY Variables

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



Syntax



Semantic Rules

1. All the semantic rules for DO FOR statements which are given in Section 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/SM 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/SM 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/SM identifier without the NAME attribute).

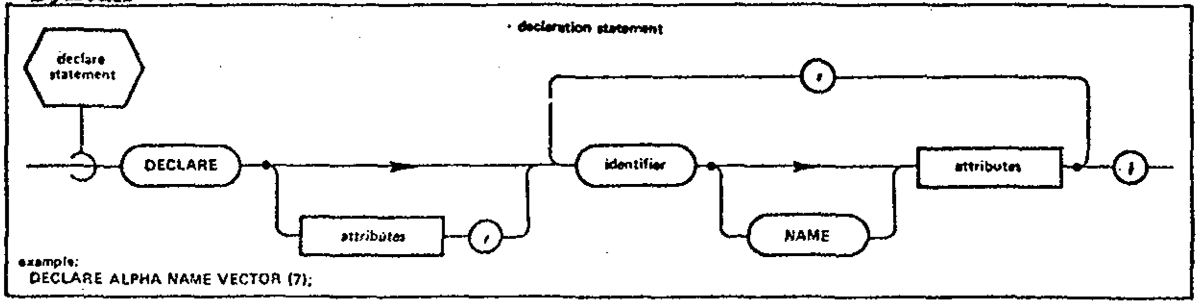
11.4.1 Identifiers with the NAME Attribute

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

INTEGER
SCALAR
VECTOR
MATRIX
BIT
BOOLEAN
CHARACTER
EVENT
STRUCTURE
DCW
TASK
PROCEDURE
FUNCTION

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



example:
DECLARE ALPHA NAME VECTOR (7);

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 specifications is excluded from delcarations 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 the same structure template.

Examples of data NAME variables

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

Z may point to Y but not X

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.

General Semantic Rules

1. The following < attribute> s 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/SM block.

   - DENSE/ALIGNED 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.

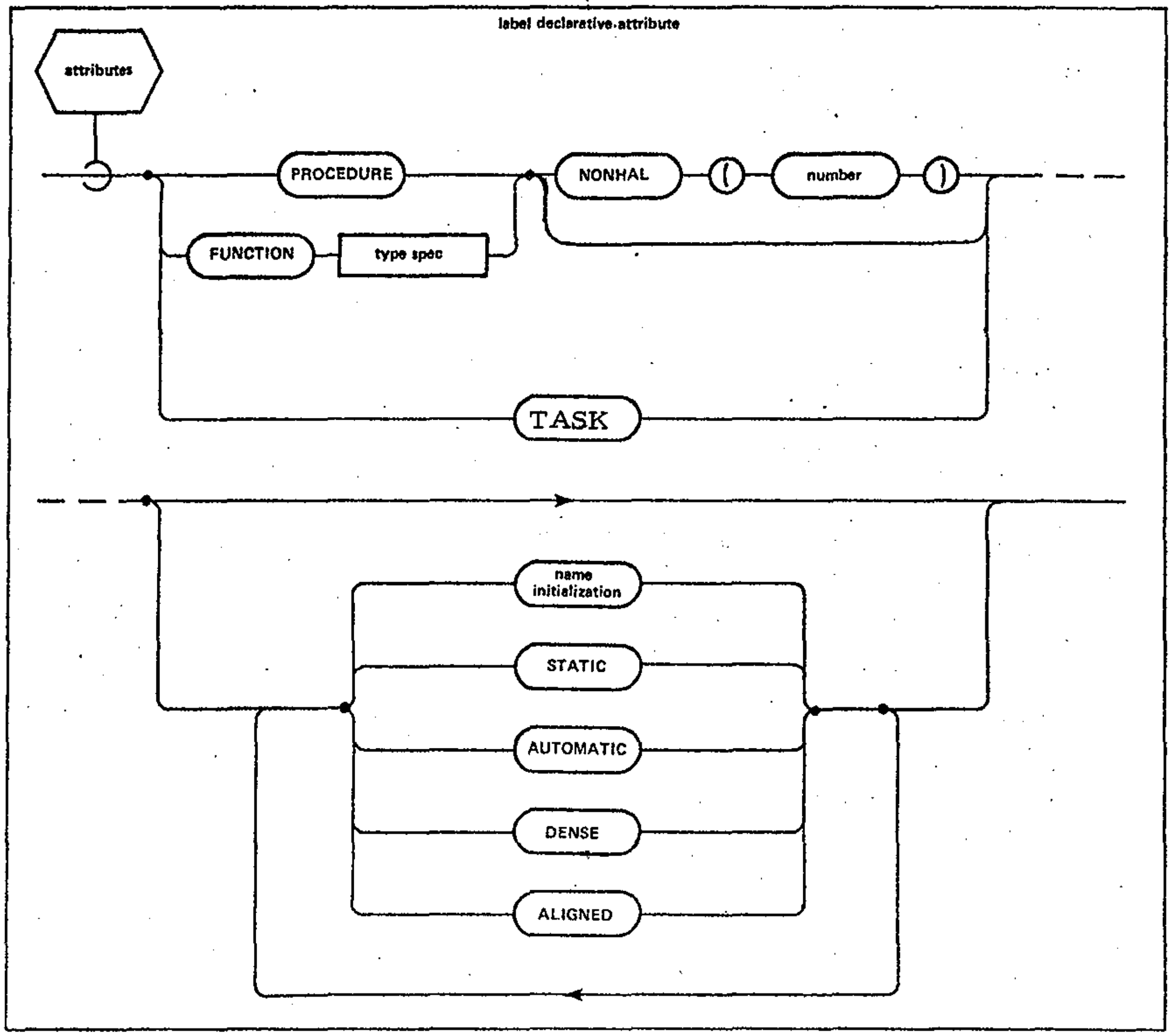
3. There must still be consistency between declared type, attributes, and factored attributes just as is the case for ordinary identifiers as described in Section 4 of this Specification.

   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
```

The Label Declarative Attributes available for use in declaring NAME identifiers which point to HAL/SM block forms have been extended to include the TASK keyword. The following syntax diagram is a modification of the Label Declarative Attributes diagram in Section 4.6.

Syntax



Semantic Rules: Label NAME Identifiers

1. An <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 PROCEDURE type:

   o Procedure NAME variables may only point to procedures defined external to the compilation and therefore for which there exists a block template in the same <unit of compilation> .

   o Only external procedures without input or ASSIGN parameters may be the objects of NAME <identifiers> s.

   o The NONHAL attribute is not allowed for NAME identifiers of the PROCEDURE type.

3. The following rules apply to NAME <identifiers> of the FUNCTION type:

   o Function NAME variables may only point to functions defined external to the compilation and therefore for which there exists a block template in the same <unit of compilation> .

   o Only external functions without input parameters may be the objects of NAME identifiers.

   o The NONHAL attribute is not allowed for NAME identifiers of the FUNCTION type.

4. The following rules apply to NAME <identifiers> of the TASK type:

   o The NAME <identifier> of a task type always points to a TASK block.

   o The only form of TASK label declarations allowed are those with the NAME attribute.

   o The task NAME <identifier> must always point to an external TASK block name; therefore a block template is required for each TASK which may be referenced by a NAME value.

Example of label NAME identifiers:

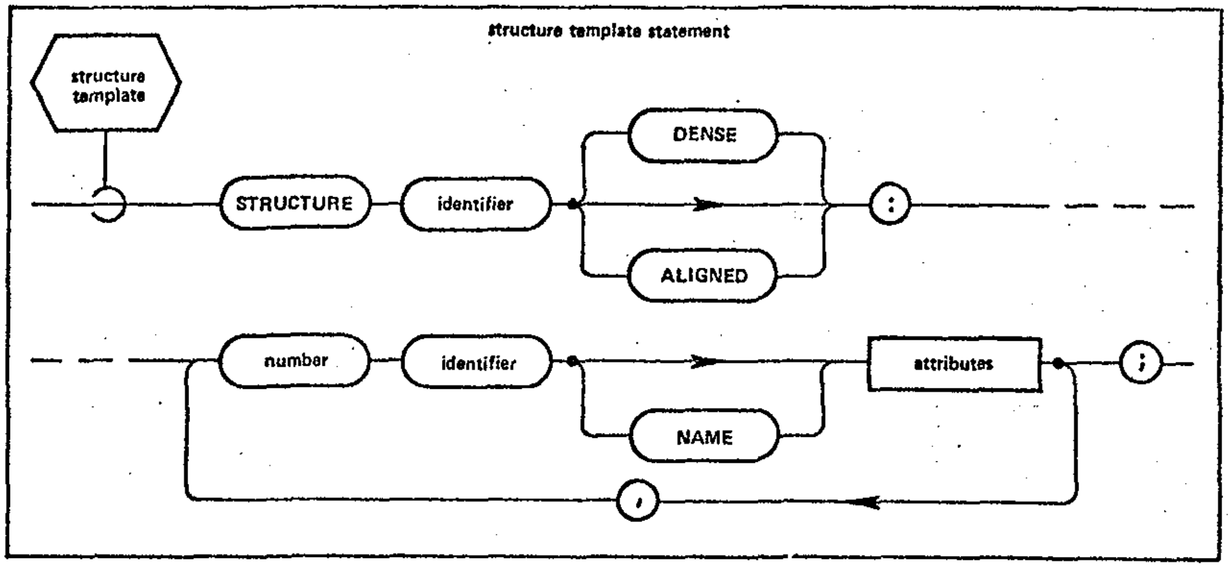
```
F:    EXTERNAL FUNCTION;
      :
      CLOSE
F1:   TASK;
      DECLARE PSI NAME FUNCTION;
      :
F2:   FUNCTION;
      :
      CLOSE;
```

PSI may point to F but not F2

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



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 task procedure and function) may appear as structure terminals. Note that the NAME of an EVENT may appear in a structure even though the EVENT itself may not.

Semantic Rules: Nested Structure Template References

1. Nested structure template references are special instances of structure terminals. The manner of their incorporation into structure template definitions is as 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:

   o Specification of multiple copies is still not permitted.

   o 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.

Examples of structure NAME identifiers:

```
STRUCTURE A:
   1 X NAME TASK,
   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.

11.4.3 Declarations of Temporaries

No identifier declared in a TEMPORARY statement may possess the NAME attribute. No such identifier of structure type may have a template which contains one or more structure terminals bearing the NAME attribute.

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/SM 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/SM 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 PROCEDURE;
Q: PROCEDURE;
    .
    .
    .
    CLOSE;
    .
    .
```

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

CALL P;	Means call 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.

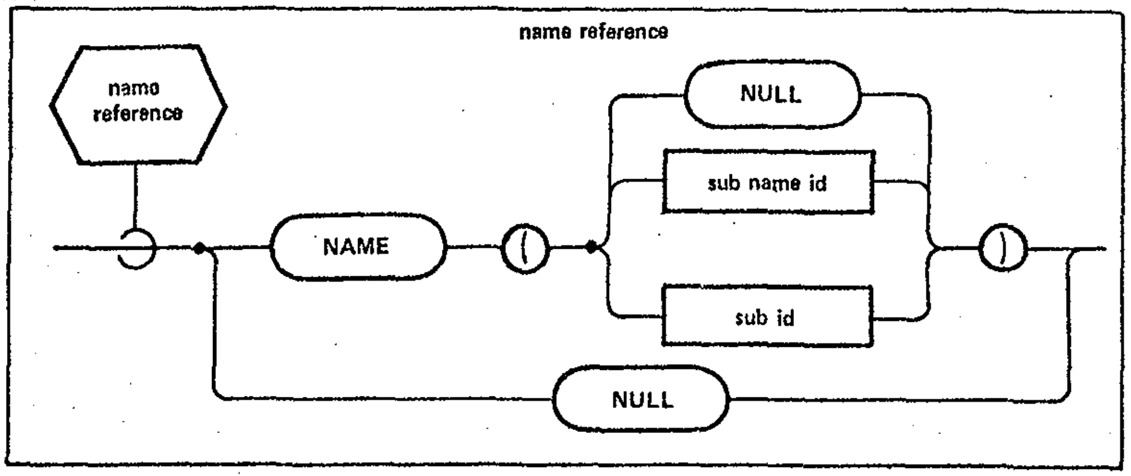
A special construct to be described in Sections 11.4.5 and 11.4.6 is required to reference or change the value of a NAME identifier (as opposed to referencing or changing the value to which it points).

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/SM identifiers. The locations or 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



Semantic Rules

1. A <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. A <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:

   o No component subscripting is allowed for bit and character types.

   o If component subscripting is present, <sub id> or < sub name id> must be subscripted so as to yield a single (unarrayed) element.

   o 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.

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 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:

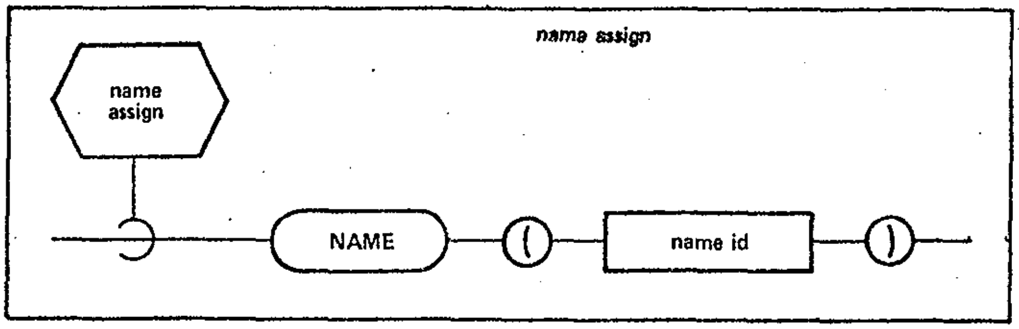
```
DECLARE X  SCALAR,
        V  VECTOR (3)
        NX NAME SCALAR
        NV NAME VECTOR (3);
   .
   .
   .
```

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.

11.4.6 Changing NAME Values

The value of a 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:

<u>Syntax</u>



Semantic Rules

1. A <name id> specifies any NAME identifier except an input parameter, whose NAME value is to be changed. A <name id> may not be subscripted (except as noted in Section 11.4.6).

Example:

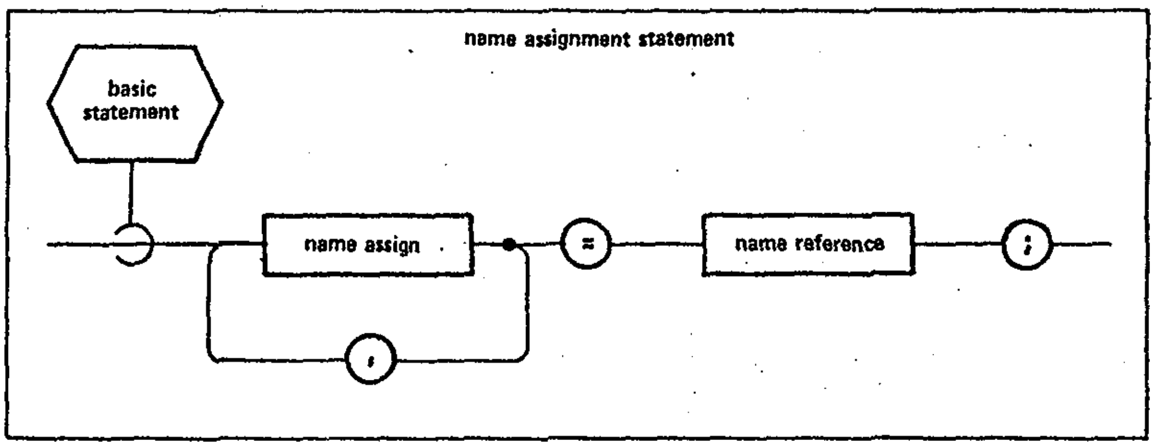
DECLARE X NAME MATRIX;

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

11.4.7 NAME Assignment Statements

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

Syntax



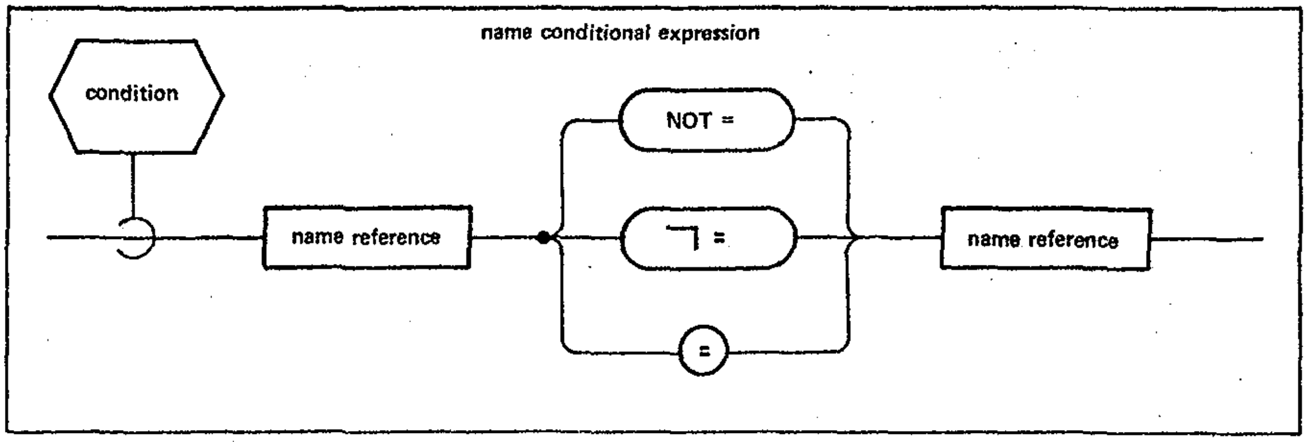
Semantic Rules

1. The <name reference> and <name assign> 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



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 arguments whose <attributes> are compatible in the sense described in Section 11.4.1.

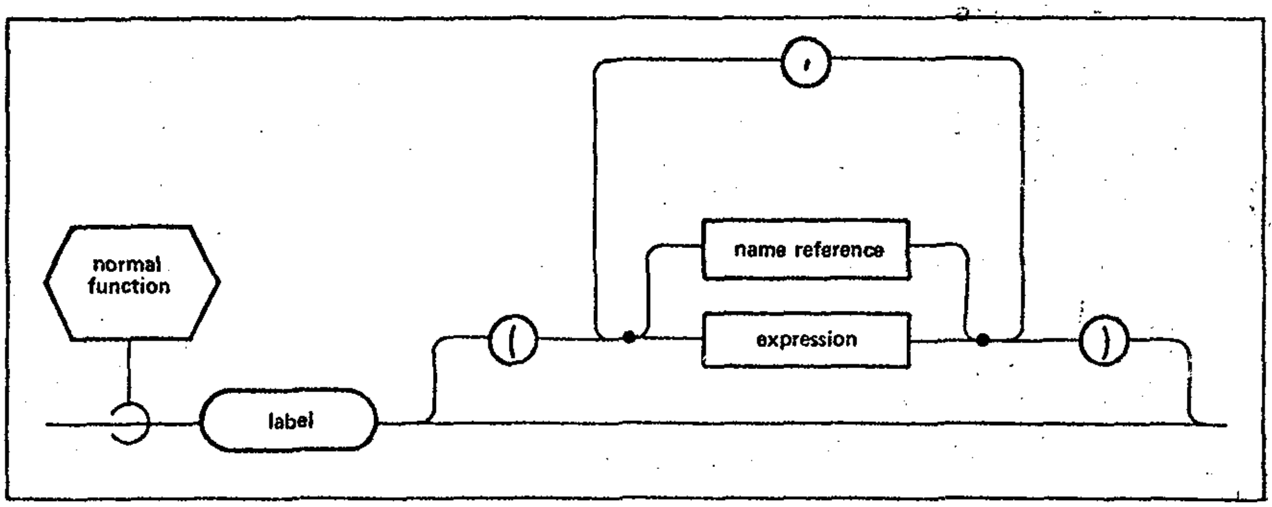
Examples:

```
        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 points nowhere,
                                    then return.
```

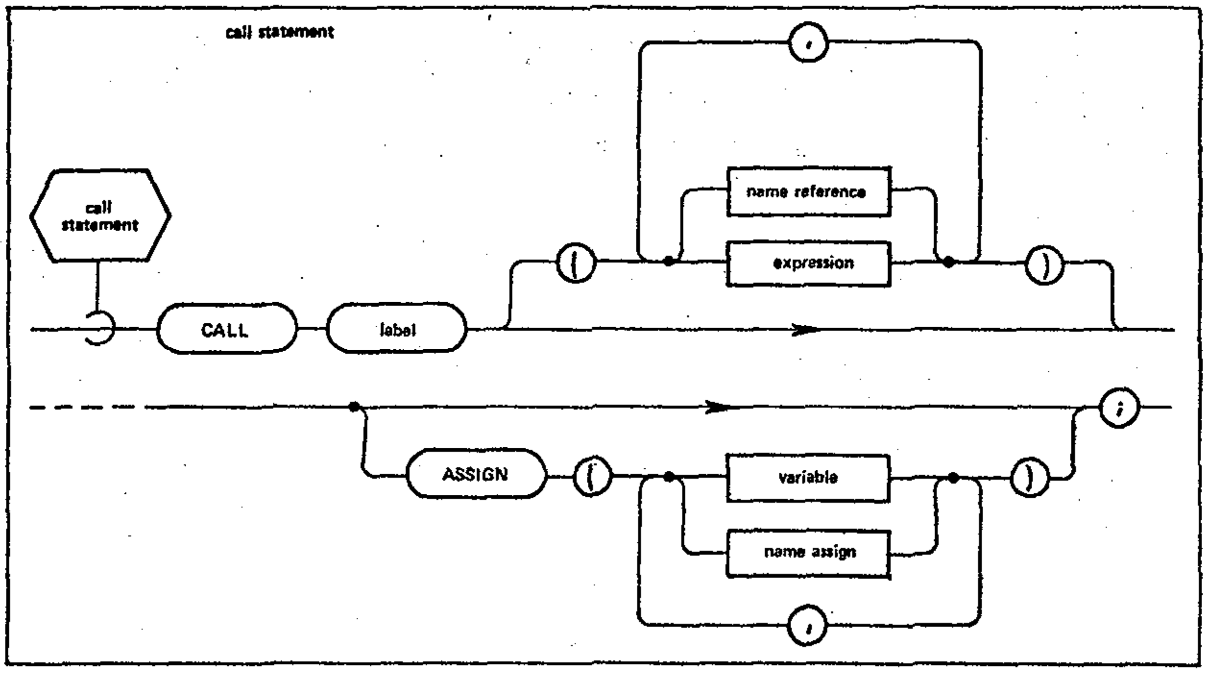
11.4.9 Argument Passage Considerations

NAME values may be passed into procedures 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



Syntax



ORIGINAL PAGE IS
OF POOR QUALITY

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.

3. If the argument of the procedure or function invocation is not a <name reference> then the corresponding formal parameter must not have the NAME attribute.

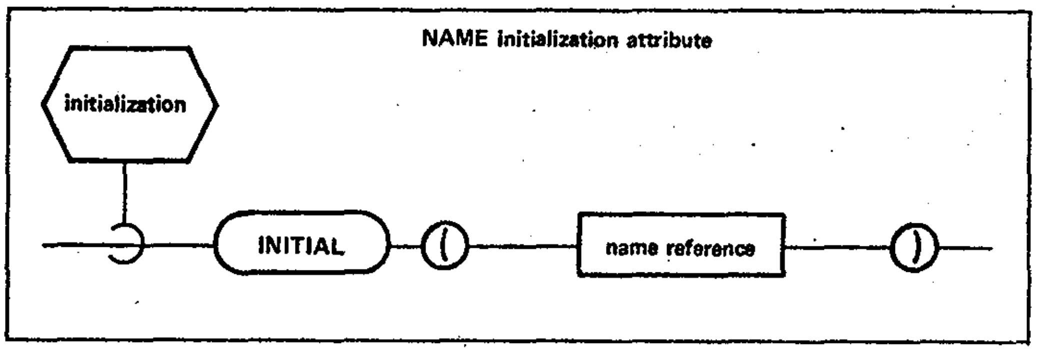
Examples:

```
DECLARE X1 SCALAR,
        X2 NAME SCALER;
•
•
•
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
    :                      parameter
CLOSE;
•
•
•
NAME(X2) = NAME(X1);
CALL P (NAME(X1), X1) ASSIGN(NAME(X2), X1);
```

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



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.

2. Subscripts are illegal in NAME initialization.

3. Uninitialized NAME identifiers will have a NULL NAME value until the first NAME assignment.

4. 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 declaration 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:

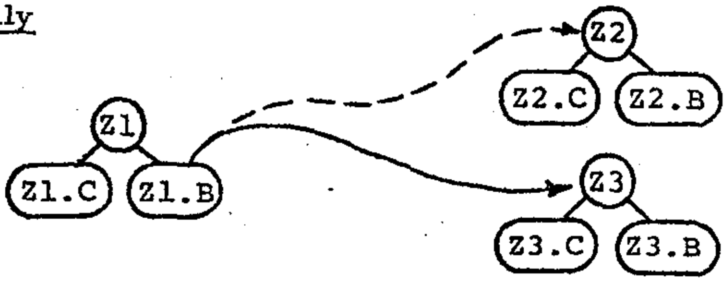
```
        NAME (Z1.B) = NAME (Z2);
```

If this is done then it is legal to specify Z1.B.C as a qualified structure terminal name. 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.

<u>Pictorially</u>

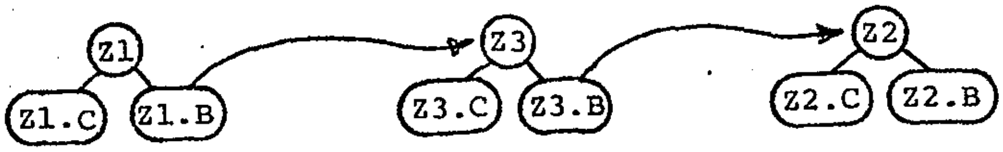


Now Z1.B.B is itself in turn a NAME identifier of A-structure type, so that if the NAME assignment

```
        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.

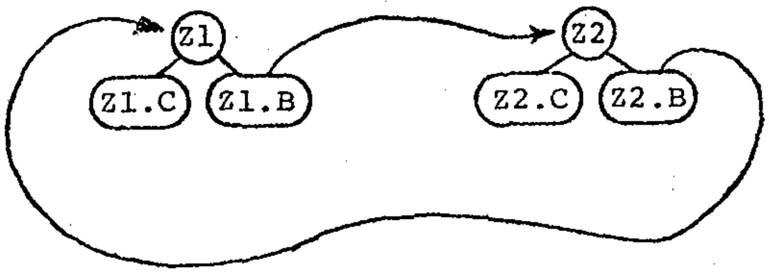
<u>Pictorially</u>



Clearly this implicit dereferencing in qualified names 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:



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

<u>Implications of Subscripting Structure Terminals</u>

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)

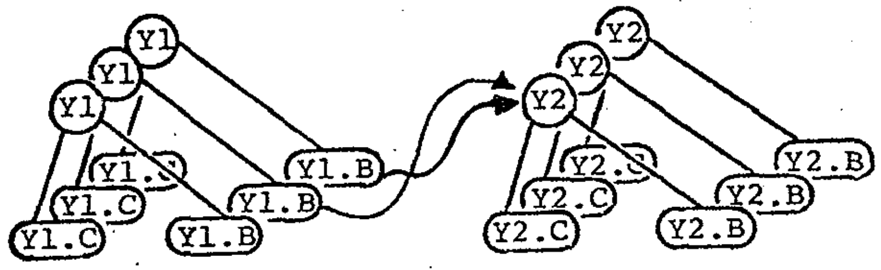
Note that now Y1.B is a NAME identifier of A-structure type with 3 copies. One or more copies of it may therefore be assigned a NAME-value at one time. For example:

$NAME\ (Y1.B_{2\ AT\ 2}) = NAME\ (Y2_{2\ AT\ 1})\ ;$

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



Notice that in the above NAME assignment a subscripted <name id> appears as argument of the left-hand side NAME pseudo-function. Subscripts so appearing are legal only if they can 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 ($Y1.B.B_2$) = NAME ($Y3_1$);

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 in this case.

<u>Manipulating Structures Containing Name Terminals</u>

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:

- o 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;
     .
     .
     .
```

ALPHA$_4$ = 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.

- For sequential I/O Cperations, 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.

BIBLIOGRAPHY

1. 'The Programming Language HAL - A Specification' Document #MSC-01846, Intermetrics, Inc., June, 1971.

2. 'HAL/S Language Specification,' Document #IR-6104, Intermetrics, Inc., June 15, 1974.