

HAL/S

LANGUAGE SPECIFICATION

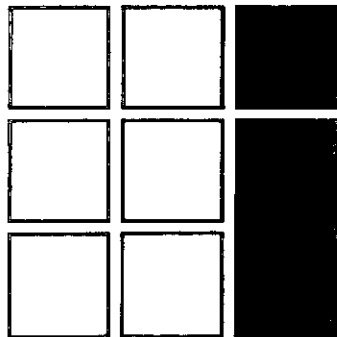
(NASA-CR-140279) HAL/S LANGUAGE
SPECIFICATION (Intermetrics, Inc.)
HC

281 p
CSCI 09B

N75-10101

Unclass
48784

G3/08



INTERMETRICS

CR 140279

UPDATE SHEET

Enclosed with each update package you receive is a new update sheeting listing the new Version number, the affected pages, and the date of the update.

As you receive each update, replace the old update sheet with the new one. It is important that you refer to the most recent Version number on the update sheet whenever you correspond with Intermetrics concerning this document.

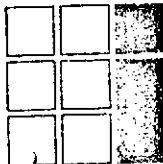
VERSION	AFFECTED MATERIAL	DATE
IR-61-8	<p>The following pages have been updated or added:</p> <ol style="list-style-type: none">1) Title Page2) Approval Page3) Page 4-10/4-10.14) Page 4-19/4-205) Page 5-9/5-106) Page 6-5/6-67) Page 6-31/6-328) Page 7-23/7-249) Page 8-3/8-410) Page 8-5/8-611) Page 11-3/11-412) Page 11-5/11-613) Page 11-19/11-2014) Page 11-21/11-2215) Page 11-29/11-3016) Page 11-31/11-3217) Page 11-41/11-4218) Page A-5 thru A-1119) Page B-1/blank page20) Page C-7/C-821) Page G-1/G-222) Page I-1/blank page23) Remove Bibliography and Section Divider24) New Index, Page I-1 thru I-29	6/16/76

HAL/S
LANGUAGE
SPECIFICATION

VERSION IR-61-8

16 JUNE 1976

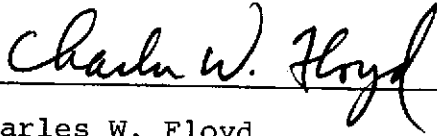
PREPARED BY



INTERMETRICS

HAL/S LANGUAGE SPECIFICATION

NASA APPROVAL:



Charles W. Floyd
Head, Systems Software Section



John R. Garman
Deputy Chief, Avionics Software Branch



Richard P. Parten
Chief, Spacecraft Software Division

INTERMETRICS APPROVAL:



Daniel J. Lickly
Head, HAL Language Compiler Department



Dr. F.H. Martin
Shuttle Program Manager

This document was originally prepared by Intermetrics Incorporated in accordance with NASA Contract NAS9-13864. It has been subsequently modified by Intermetrics under a subcontract to IBM Corporation in accordance with NASA contract NAS 9-14444.

PREFACE

The HAL Programming Language has been developed by the staff of Intermetrics, Inc. based on many years of experience in producing software for aerospace applications.

HAL accomplishes three significant objectives:

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

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

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

The HAL/S Language Specification was prepared by the staff of Intermetrics, Inc. under the direction of Dr. Philip Newbold, the document's principal author. Contributions were also made by Arra Avakian, Carl Helmers, Andy Johnson, Ron Kole, Dan Lickly, Fred Martin, Joe Saponaro, and Woody Vandever.

Editorial assistance was provided by Lee Hotz, and the typescript was prepared by Valerie Cripps.

TABLE OF CONTENTS

	<u>PAGE</u>
1. INTRODUCTION	1-1
1.1 Purpose of the Document	1-1
1.2 Review of the Language	1-1
1.3 Outline of the Document	1-3
2. SYNTAX DIAGRAMS AND HAL/S PRIMITIVES	2-1
2.1 The HAL/S Syntax Diagram	2-2
2.2 The HAL/S Character Set	2-4
2.3 HAL/S Primitives	2-5
2.3.1 <i>Reserved Words</i>	2-6
2.3.2 <i>Identifiers</i>	2-7
2.3.3 <i>Literals</i>	2-8
2.4 One- and Two-Dimensional Source Formats	2-11
2.5 Comments and Blanks in the Source Text	2-13
3. HAL/S BLOCK STRUCTURE AND ORGANIZATION (Diags. 1-10)	3-1
3.1 The Unit of Compilation	3-2
3.2 The PROGRAM Block	3-4
3.3 PROCEDURE, FUNCTION, and TASK Blocks	3-6
3.4 The UPDATE Block	3-8
3.5 The COMPOOL Block	3-10
3.6 Block Templates	3-11
3.7 Block Delimiting Statements	3-13

3.7.1	<i>Simple Header Statements</i>	3-14
3.7.2	<i>The Procedure Header Statement</i>	3-15
3.7.3	<i>The Function Header Statement</i>	3-17
3.7.4	<i>The CLOSE Statement</i>	3-19
3.8	Name Scope Rules	3-20
4.	DATA AND OTHER DECLARATIONS (Diags. 11-18)	4-1
4.1	The Declare Group	4-3
4.2	The REPLACE Statement	4-4
4.2.1	<i>Form of REPLACE Statement</i>	4-4
4.2.2	<i>Referencing REPLACE Statements</i>	4-6
4.3	The Structure Template	4-8
4.4	The DECLARE Statement	4-12
4.5	Data Declarative Attributes	4-13
4.6	Label Declarative Attributes	4-18
4.7	Type Specification	4-19
4.8	Initialization	4-23
5.	DATA REFERENCING CONSIDERATIONS	5-1
5.1	Referencing Simple Variables	5-2
5.2	Referencing Structures	5-3
5.3	Subscripting	5-5
5.3.1	<i>Classes of Subscripting</i>	5-7
5.3.2	<i>The General Form of Subscripting</i>	5-11
5.3.3	<i>Structure Subscripting</i>	5-13
5.3.4	<i>Array Subscripting</i>	5-14

5.3.5	<i>Component Subscripting</i>	5-15
5.4	The Property of Arrayness	5-17
5.4.1	<i>Arrayness of Subscript Expressions</i>	5-18
5.5	The Natural Sequence of Data Elements	5-20
6.	DATA MANIPULATION AND EXPRESSIONS	6-1
6.1	Regular Expressions	6-2
6.1.1	<i>Arithmetic Expressions</i>	6-3
6.1.2	<i>Bit Expressions</i>	6-7
6.1.3	<i>Character Expressions</i>	6-10
6.1.4	<i>Structure Expressions</i>	6-12
6.2	Conditional Expressions	6-13
6.2.1	<i>Arithmetic Comparisons</i>	6-15
6.2.2	<i>Bit Comparisons</i>	6-17
6.2.3	<i>Character Comparisons</i>	6-18
6.2.4	<i>Structure Comparisons</i>	6-19
6.2.5	<i>Comparisons between Arrayed Operands</i>	6-20
6.3	Event Expressions	6-21
6.4	Normal Functions	6-23
6.5	Explicit Type Conversions	6-26
6.5.1	<i>Arithmetic Conversion Functions</i>	6-27
6.5.2	<i>The Bit Conversion Function</i>	6-31
6.5.3	<i>The Character Conversion Function</i>	6-33
6.5.4	<i>The SUBBIT Pseudo-variable</i>	6-35
6.5.5	<i>Summary of Argument Types</i>	6-37
6.6	Explicit Precision Conversion	6-38

7.	EXECUTABLE STATEMENTS (Diags. 44-56)	7-1
7.1	Basic Statements	7-2
7.2	The IF Statement	7-3
7.3	The Assignment Statement	7-5
7.4	The CALL Statement	7-9
7.5	The RETURN Statement	7-12
7.6	The DO...END Statement Group	7-14
	7.6.1 <i>The Simple DO Statement</i>	7-15
	7.6.2 <i>The DO CASE Statement</i>	7-16
	7.6.3 <i>The DO WHILE and UNTIL Statements</i>	7-17
	7.6.4 <i>The Discrete DO FOR Statement</i>	7-19
	7.6.5 <i>The Iterative DO FOR Statement</i>	7-21
	7.6.6 <i>The END Statement</i>	7-23
7.7	Other Basic Statements	7-24
8.	REAL TIME CONTROL (Diags. 57-62)	8-1
8.1	Real Time Processes and the RTE	8-2
8.2	Timing Considerations	8-3
8.3	The SCHEDULE Statement	8-4
8.4	The CANCEL Statement	8-9
8.5	The TERMINATE Statement	8-11
8.6	The WAIT Statement	8-12
8.7	The UPDATE PRIORITY Statement	8-14
8.8	Events and the SIGNAL Statement	8-15
8.9	Process-events	8-18
8.10	Data Sharing and the UPDATE Block	8-19

9.	ERROR RECOVERY AND CONTROL	9-1
9.1	The ON ERROR Statement	9-2
9.2	The SEND ERROR Statement	9-7
10.	INPUT/OUTPUT STATEMENTS	10-1
10.1	Sequential I/O Statements	10-2
10.1.1	<i>The READ and READALL Statements</i>	10-3
10.1.2	<i>The WRITE Statement</i>	10-6
10.1.3	<i>I/O Control Functions</i>	10-8
10.2	Random Access I/O and the FILE Statement	10-10
11.	SYSTEMS LANGUAGE FEATURES	11-1
11.1	Introduction	11-1
11.2	Program Organization Features	11-1
11.2.1	<i>Inline Function Blocks</i>	11-2
11.2.2	<i>%-macro References</i>	11-4
11.2.3	<i>Operand Reference Invocations</i>	11-6
11.2.4	<i>The %-Macro Call Statement</i>	11-11
11.3	Temporary Variables	11-12
11.3.1	<i>Regular TEMPORARY Variables</i>	11-12
11.3.2	<i>Loop TEMPORARY Variables</i>	11-14
11.4	The NAME Facility	11-16
11.4.1	<i>Identifiers with the NAME Attribute</i>	11-16
11.4.2	<i>The NAME Attribute in Structure Templates</i>	11-22
11.4.3	<i>Declarations of Temporaries</i>	11-24
11.4.4	<i>The 'Dereferenced' Use of Simple NAME Identifiers</i>	11-25

11.4.5	<i>Referencing NAME Values</i>	11-26
11.4.6	<i>Changing NAME Values</i>	11-29
11.4.7	<i>NAME Assignment Statements</i>	11-29
11.4.8	<i>NAME Value Comparisons</i>	11-30
11.4.9	<i>Argument Passage Considerations</i>	11-31
11.4.10	<i>Initialization</i>	11-33
11.4.11	<i>Notes on NAME Data Structures</i>	11-34

APPENDICES

A.	SYNTAX DIAGRAM SUMMARIES	A-1
B.	HAL/S KEYWORDS	B-1
C.	BUILT-IN FUNCTIONS	C-1
D.	STANDARD CONVERSION FORMATS	D-1
E.	STANDARD EXTERNAL FORMATS	E-1
F.	COMPILE-TIME COMPUTATIONS	F-1
G.	WORKING GRAMMAR	G-1
H.	SUMMARY OF OPERATORS	H-1

BIBLIOGRAPHY

INDEX

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/S is a dialect of the original HAL language previously developed by Intermetrics [1]. Changes have been incorporated to simplify syntax, curb excessive generality, or facilitate flight code emission.

1.1 Purpose of the Document.

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

1.2 Review of the Language.

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

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

DATA TYPES AND COMPUTATIONS

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

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

REAL TIME CONTROL

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

ERROR RECOVERY

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

SYSTEM LANGUAGE

HAL/S contains a number of features especially designed to facilitate its 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/S is a block oriented language in that blocks of code may be established with locally defined variables that

are not visible from outside the block. Separately compiled program blocks can be executed together and communicate through one or more centrally managed and highly visible data pools. In a real time environment, HAL/S couples these precautions with locking mechanisms preventing the uncontrolled usage of sensitive data or areas of code.

1.3 Outline of the Document.

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

The global structure of HAL/S is presented in Section 3. Data declaration and referencing are presented in Sections 4 and 5 respectively. Section 6 is 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/S error recovery system and Section 10 the HAL/S I/O capability.

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

2. SYNTAX DIAGRAMS AND HAL/S PRIMITIVES

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

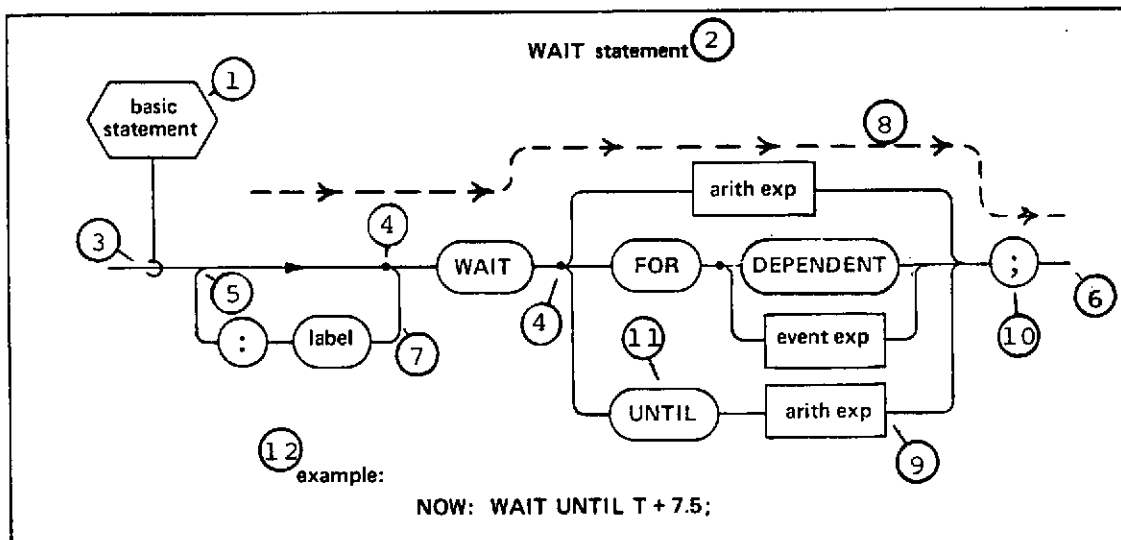
This Section has two main purposes: to explain how to read syntax diagrams, and to provide definitions of the HAL/S primitives. Various aspects of HAL source text which impact upon the meaning of the diagrams are also discussed briefly.

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

2.1 The HAL/S Syntax Diagram.

Syntax diagrams are, essentially, flow diagrams representing the formal grammar of a language. By tracing the paths on a diagram, various examples of the language construct it represents may be created. In this Specification, the Syntax Diagrams, together with the associated Semantic Rules, provide a complete and unambiguous definition of the HAL/S Language. The syntax diagrams are, however, not meant to be viewed as constituting a "working" grammar (that is, 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 <arith exp>," (see Rule 7.)
6. Boxes with squared corners, such as ⑨, represent syntactical terms defined in other diagrams. Boxes with circular ends, such as ⑪, represent HAL/S primitives. Circular boxes, such as ⑩, contain special characters (see Section 2.2).
7. 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 <arith exp>. Upper case names are reserved words of the language.
8. The example given at ⑫ is an example of HAL/S code which may be generated by applying the syntax diagram (since some boxes, such as ⑨ for example, are defined in other syntax diagrams, reference to them may be necessary to complete the generative process).

2.3 HAL/S Primitives.

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

2.3.1 *Reserved Words*

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

2.3.2 Identifiers.

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

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

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

<label>	
<process-event name>	
<§ var name>	where § ≡
<template name>	{
	arith (arithmetic)
	char (character)
	bit
	event
	structure
	}

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

2.3.3 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.

RULES FOR ARITHMETIC LITERALS:

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

2. The generic form of a <number> is

$$^{\pm} \text{ddddddd.dddddddd} \langle \text{exponents} \rangle$$

where d = decimal digit.

Any number of decimal digits to an implementation dependent maximum, including none, may appear before or after the decimal point. The sign and 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>	~ 2<power>
E<power>	~ 10<power>
H<power>	~ 16<power>

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

examples:

0.123E16B-3
45.9
-4

RULES FOR BIT LITERALS:

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

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

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

3. The following abbreviated forms are allowed:

TRUE \equiv ON \equiv BIN'1'

FALSE \equiv OFF \equiv 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

'cccccc'

CHAR <repetition> 'cccccc'

where c is any character in the HAL/S extended character set. The <repetition> consists of a parenthesized positive integer literal. It indicates how many times the following string is to be used in creating the value. The number of characters lies between zero and an implementation dependent maximum.

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

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

examples:

' '

'ONE TWO THREE'

'DOG' 'S'

'ABϕAD' }

'ABϕϕAD' }

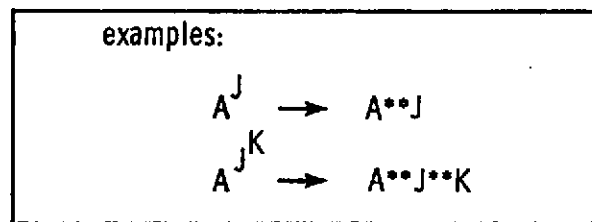
The implication is that ϕA
and ϕϕA have been defined
as alternate characters.

2.4 One- and Two-Dimensional Source Formats.

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

RULES FOR EXPONENTS:

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



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

RULES FOR SUBSCRIPTS:

1. In the syntax diagrams, 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:

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

3. For certain simple forms of subscript, the parentheses may be omitted. These forms are:
- a single <number>
 - a single <arith var name> (see Section 5.3).

example:

$$A_J \rightarrow A\$J$$

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

2.5 Comments and Blanks in the Source Text.

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

- between two primitives;
- between two special characters;
- between a primitive and a special character.

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

Comments may be imbedded within HAL 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.

3. HAL/S BLOCK STRUCTURE AND ORGANIZATION

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

At run time, an arbitrary number of object modules are combined to form an executable "program complex"¹. Generally, a program complex contains three different types of object modules:

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

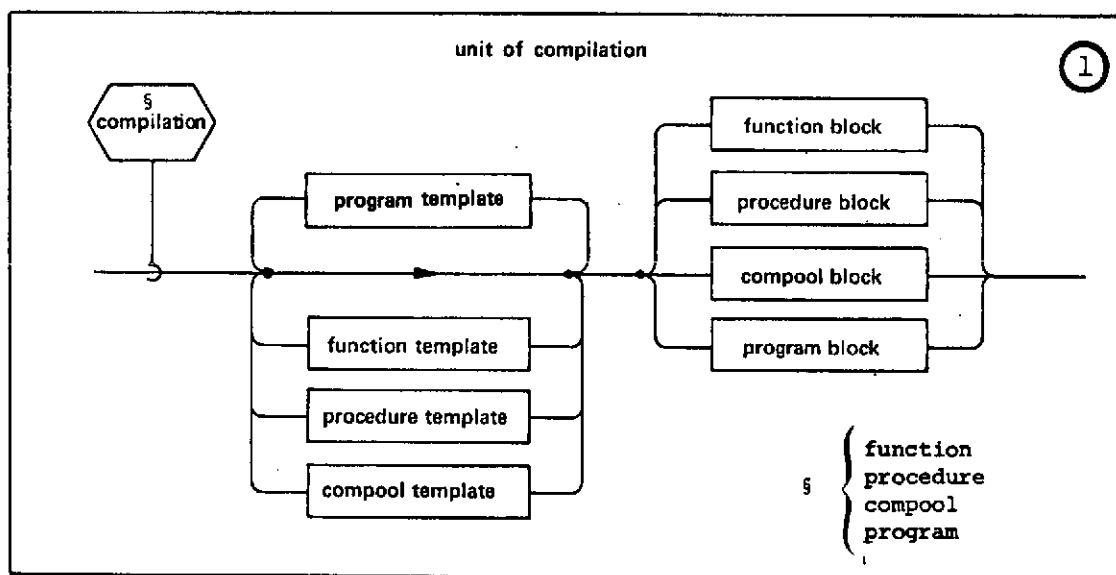
Each module originates from a unit of compilation of corresponding type.

¹ A program complex is executable within the framework of an executive operating system, and a run time utility library.

3.1 The Unit of Compilation.

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

SYNTAX:



SEMANTIC RULES:

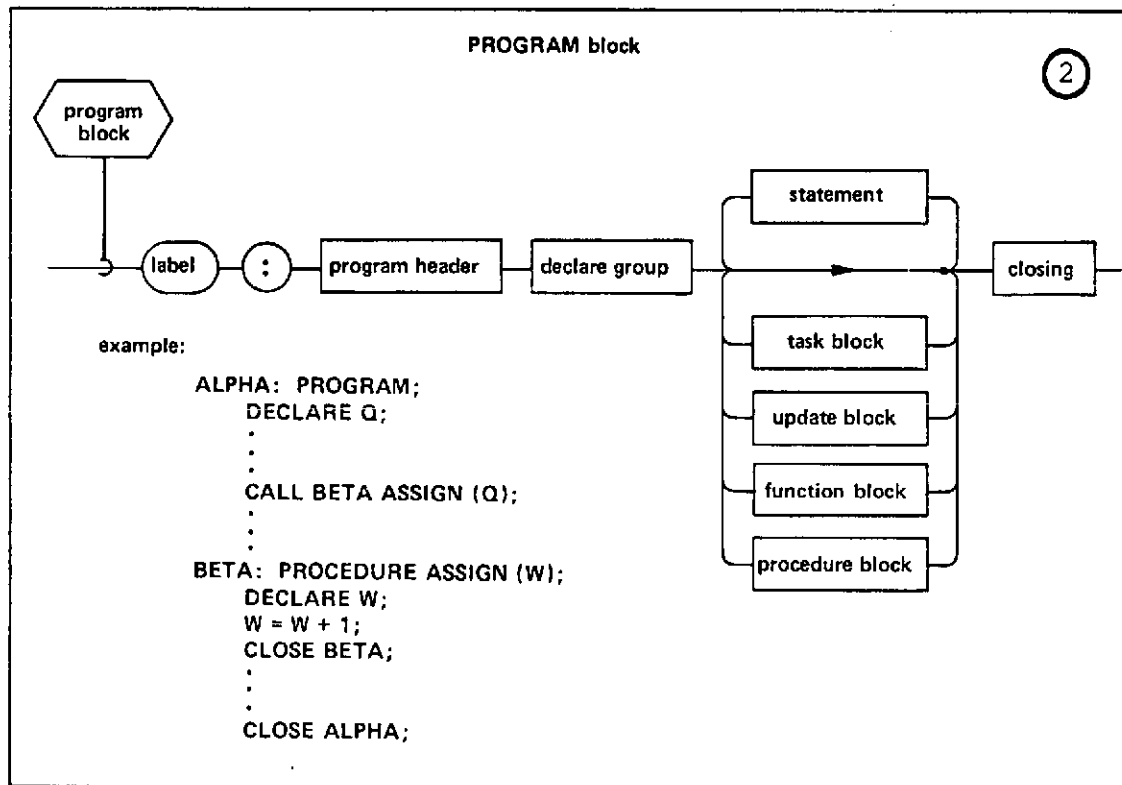
1. A program <compilation> is one containing a <program block>. Its object module in the program complex may be activated by the Real Time Executive (see Section 8.), or by other means dependent on the operating system. The <program block> is described in Section 3.2.
2. A procedure or function <compilation> is one containing a <procedure block> or <function block> respectively. Its object module in the program complex is executed by being invoked by other program, procedure or function modules. Both <procedure block>s and <function block>s are described in Section 3.3.

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

3.2 The PROGRAM Block.

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

SYNTAX:



SEMANTIC RULES:

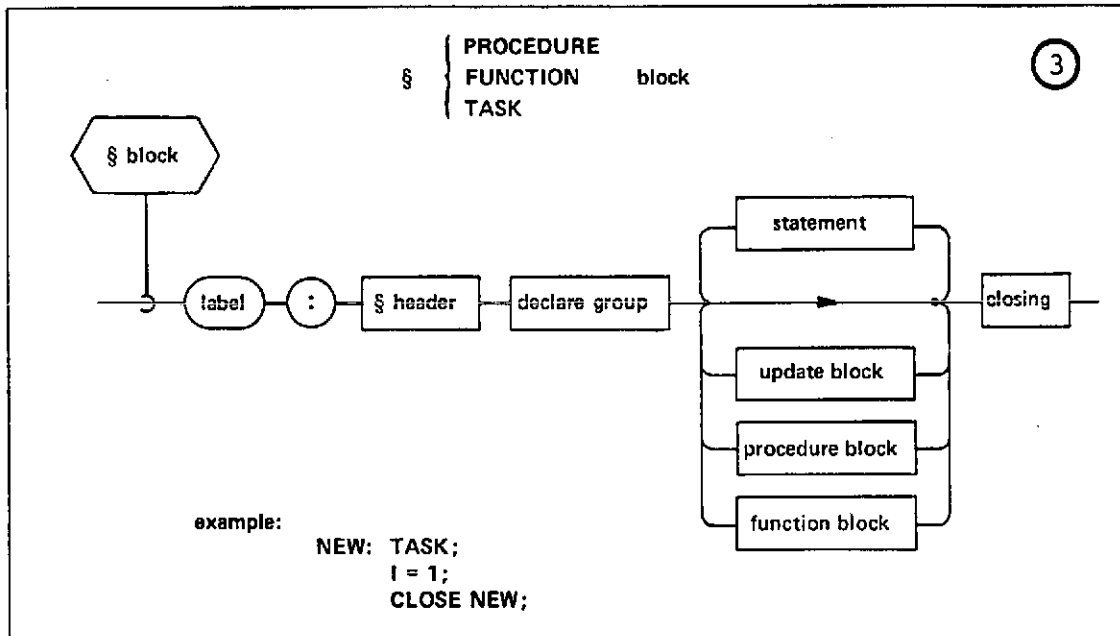
1. The name of the <program block> is given by the <label> prefacing the block.
2. The <program block> is delimited by a <program header> statement at the beginning, and a <closing> at the end. These two delimiting statements are described in Sections 3.7.1 and 3.7.4 respectively.
3. The contents of a <program block> consist of a <declare group> used to define data local to the <program block>, followed by any number of executable <statement>s.

4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement> may modify this normal sequencing in a well-defined way.
5. PROCEDURE, FUNCTION, TASK, and UPDATE blocks may appear nested within a <program block>. The blocks may be interspersed between the <statement>s of the <program block>, and with the exception of the UPDATE block are not executed in-line.
6. Execution of a <program block> is accomplished by scheduling it as a process under the control of the Real Time Executive (see Section 8.).

3.3 PROCEDURE, FUNCTION, and TASK Blocks.

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

SYNTAX:



SEMANTIC RULES:

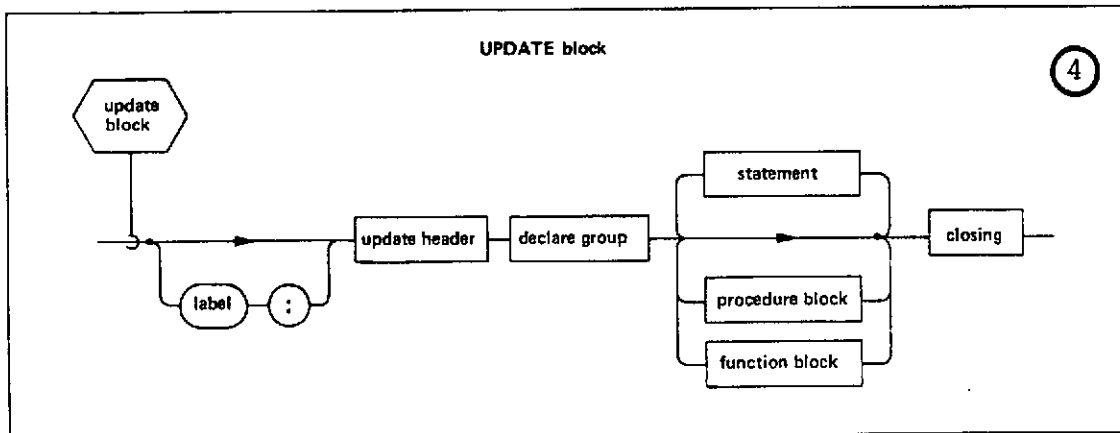
1. The name of the block is given by the <label> prefacing the block. The definition of a block label is considered to be in the scope of the outer block containing the block in question. Block names must be unique within any compilation unit.
2. The block is delimited at its beginning by a header statement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

4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement> may modify this normal sequencing in a well-defined way.
5. 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. Execution of a <task block> is invoked by scheduling it as a process under the control of the Real Time Executive (see Section 8.). Execution of a <procedure block> is invoked by the CALL statement (see Section 7.4). Execution of a <function block> is invoked by the appearance of its name in an expression (see Section 6.4).
7. 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:

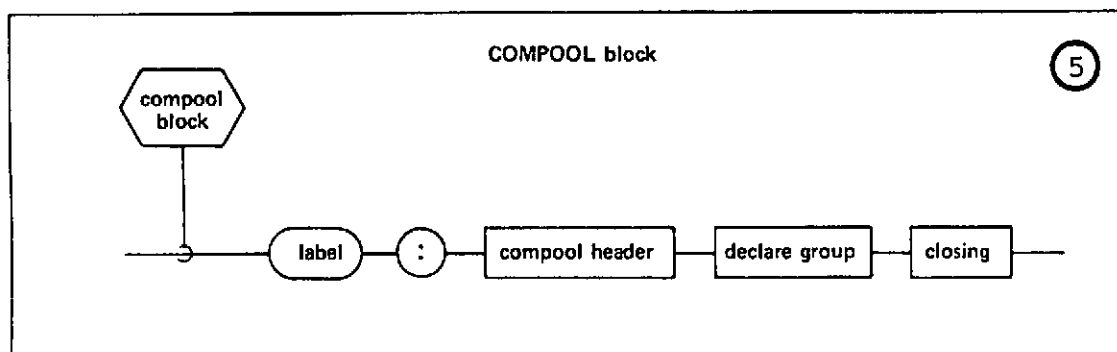
1. If present, the <label> prefacing the <update block> gives the name of the block. If <label> is absent, the <update block> is unnamed.
2. The block is delimited at its beginning by an <update header> statement, and at the end by a <closing>. The delimiting statements are described in Sections 3.7.1 and 3.7.4.
3. The contents of the block consist of a <declare group> used to declare data local to the <update block>, followed by any number of executable <statement>s.
4. The normal flow of execution of the <statement>s in the block is sequential; various types of <statement> may modify this normal sequencing in a well-defined way.
5. Only PROCEDURE and FUNCTION blocks may be nested within an <update block>. The nested blocks may appear interspersed between the <statement>s of the block, and are not executed in-line.

6. An <update block> is treated like a <statement> in that it is executed in-line. In this respect it is different from other code blocks.
7. The following <statement>s are expressly forbidden inside an <update block> in view of its special protective function:
 - I/O statements (see Section 10.);
 - invocations of <procedure block>s or <function block>s not themselves nested within the <update block>;
 - real-time programming statements, except for the SIGNAL, SET, and RESET statements (see Section 8.8).

3.5 The COMPOOL Block.

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

SYNTAX:



SEMANTIC RULES:

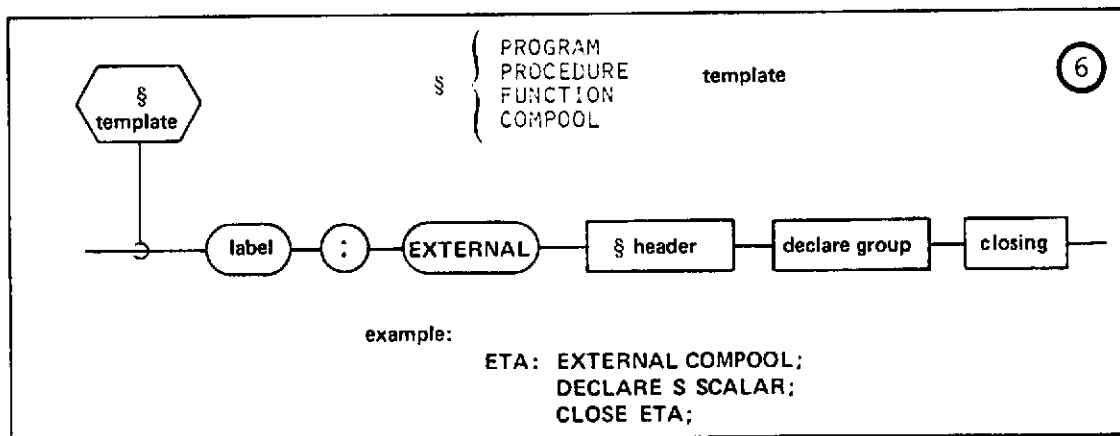
1. The name of the block is given by the <label> prefacing the block.
2. The block is delimited at its beginning by a <compool header> statement, and at its end by a <closing>. The delimiting statements are described in Sections 3.7.1 and 3.7.4.
3. The contents of the block consist merely of a <declare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

3.6 Block Templates.

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

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

SYNTAX:



SEMANTIC RULES:

1. The <label> of the template constitutes the template name. It is the same name as that of the code block to which the template corresponds.
2. The block template is delimited at its beginning by a header statement identical with the header statement of the corresponding code block, and at the end by a <closing>. The delimiting statements are described in Sections 3.7.1 through 3.7.4.

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

3.7 Block Delimiting Statements.

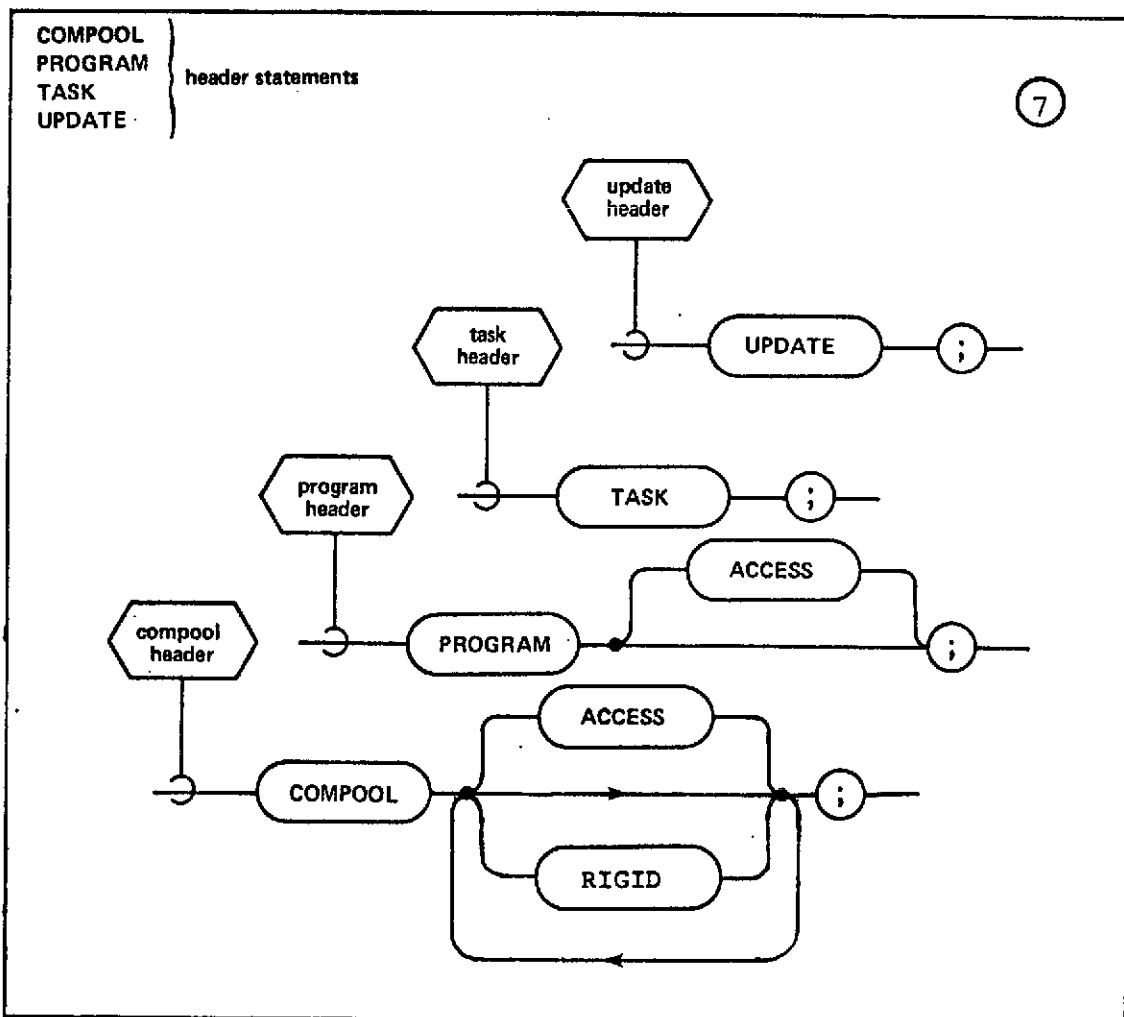
Both code blocks and block templates are delimited at the beginning by a header statement characteristic of their type, and at the end by a <closing> statement. In all code blocks except for the COMPOOL block, the header statement is the first statement of the block to be executed on entry. A COMPOOL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

124

3.7.1 Simple Header Statements

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

SYNTAX:



90

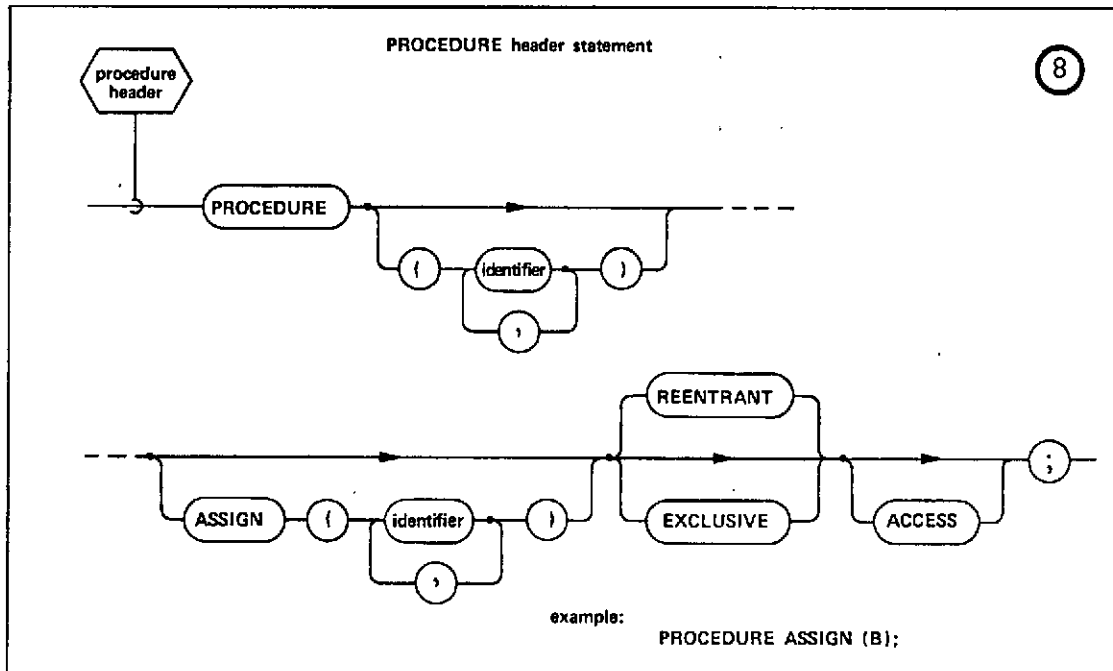
SEMANTIC RULES:

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

3.7.2 The Procedure Header Statement.

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

SYNTAX:



SEMANTIC RULES:

1. The keyword PROCEDURE identifies the start of a <procedure block>, or <procedure template>. It is optionally followed by lists of "formal parameters" which correspond to "arguments" in the invocation of the procedure by a CALL statement (see Section 7.4).
2. The <identifier>s in the list following the PROCEDURE keyword are called "input parameters" because they may not appear in any context inside the code block which may cause their values to be changed.

3. The <identifier>s in the list following the ASSIGN keyword are called "assign parameters" because they may appear in contexts inside the code block in which new values may be assigned to them. They may, of course, also appear in the same contexts as input parameters.
4. Data declarations for all formal parameters must appear in the <declare group> of the <procedure block> or <procedure template>.
5. If the <procedure header> statement specifies neither of the keywords REENTRANT or EXCLUSIVE, then only one real time process (see Section 8.) may be executing the <procedure block> at any one time; however there is no enforcing protective mechanism. If the keyword EXCLUSIVE is specified, then such a protective mechanism does exist. If an EXCLUSIVE <procedure block> is already being executed by a real time process when a second process tries to invoke it, the second process is forced into the stall state (see Section 8.) until the first has finished executing it. If the keyword REENTRANT is specified, then two or more processes may execute the <procedure block> "simultaneously".
6. The keyword REENTRANT indicates to the compiler that reentrancy is desired. However, other attributes and conditions may conflict with this overall objective. The following effects should be noted:
 - STATIC data is allocated statically and initialized statically. There is only one copy of STATIC data which must be shared by all processes simultaneously executing the block. Hence, in coding REENTRANT blocks care must be taken not to assume that STATIC variables participate in the reentrancy.
 - AUTOMATIC data is allocated dynamically and initialized dynamically. Every process simultaneously executing the block gets its own initialized copy of the data on entry into the block. In general, all local data in a REENTRANT block should be declared with the AUTOMATIC attribute.
 - Procedures and functions defined within a REENTRANT block must also possess the REENTRANT attribute if they too declare local data which is required to participate in the reentrancy.

In addition, for reentrancy to be preserved, the following rules must be observed:

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

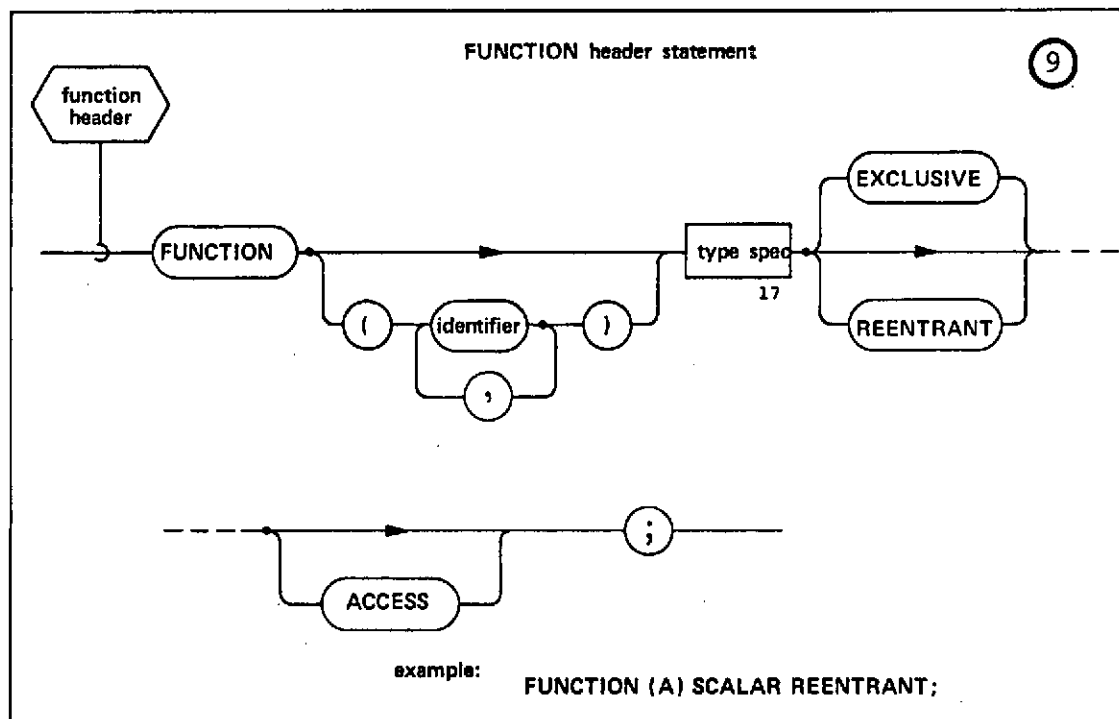
107

* Any use of update blocks and LOCK data, and of EXCLUSIVE procedure or function blocks should be carefully analyzed with respect to unfavorable interactions with REENTRANT blocks.

3.7.3 The Function Header Statement.

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

SYNTAX:



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. <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 specifies neither of the keywords REENTRANT or EXCLUSIVE, then only one real time process (see Section 8.) may be executing the <function block> at any one time; however there is no enforcing protective mechanism. If the keyword EXCLUSIVE is specified, then such a protective mechanism does exist. If an EXCLUSIVE <function block> is already being executed by a real time process when a second process tries to invoke it, the second process is forced into the stall state (see Section 8.) until the first has finished executing it. If the keyword REENTRANT is specified, then two or more processes may execute the <function block> "simultaneously".
6. The keyword REENTRANT indicates to the compiler that reentrancy is desired. However, other attributes and conditions may conflict with this overall objective. The following effects should be noted:
 - STATIC data is allocated statically and initialized statically. There is only one copy of STATIC data which must be shared by all processes simultaneously executing the block. Hence, in coding REENTRANT blocks care must be taken not to assume that STATIC variables participate in the reentrancy.
 - AUTOMATIC data is allocated dynamically and initialized dynamically. Every process simultaneously executing the block gets its own initialized copy of the data on entry into the block. In general, all local data in a REENTRANT block should be declared with the AUTOMATIC attribute.
 - Procedures and functions defined within a REENTRANT block must also possess the REENTRANT attribute if they too declare local data which is required to participate in the reentrancy.

107

In addition, for reentrancy to be preserved, the following rules must be observed:

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

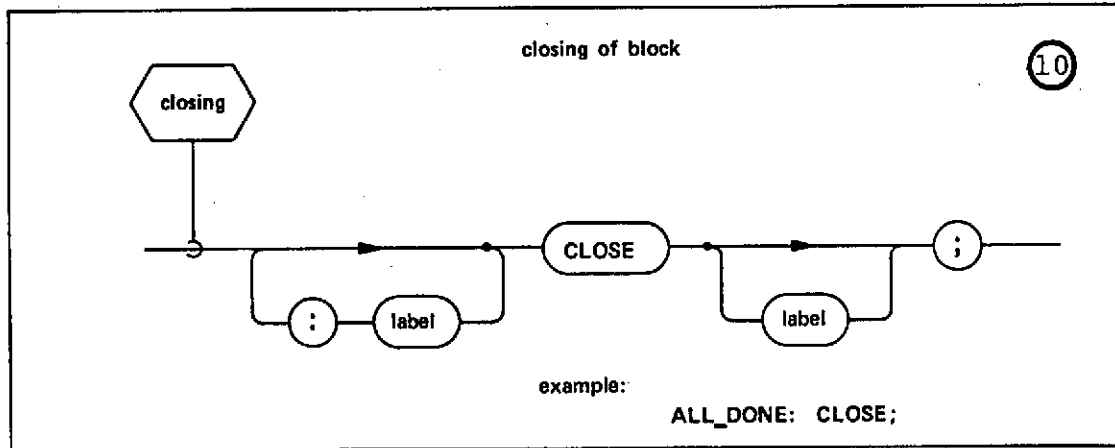
107

* Any use of update blocks and LOCK data, and of EXCLUSIVE procedure or function blocks should be carefully analyzed with respect to unfavorable interactions with REENTRANT blocks.

3.7.4 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 PROGRAM, PROCEDURE, TASK, or UPDATE block, and a run time error from a FUNCTION block. Exit from a FUNCTION block must be achieved via the RETURN statement (see Section 7.5).
3. The <closing> of a PROGRAM, PROCEDURE, FUNCTION, TASK, or UPDATE block may be labelled as if it were a <statement>. The <closing>s of COMPOOL blocks and block templates cannot be labelled.

3.8 Name Scope Rules.

By using the code blocks described, and by taking advantage of their nesting property, the modularization of HAL/S <compilation>s may be 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.

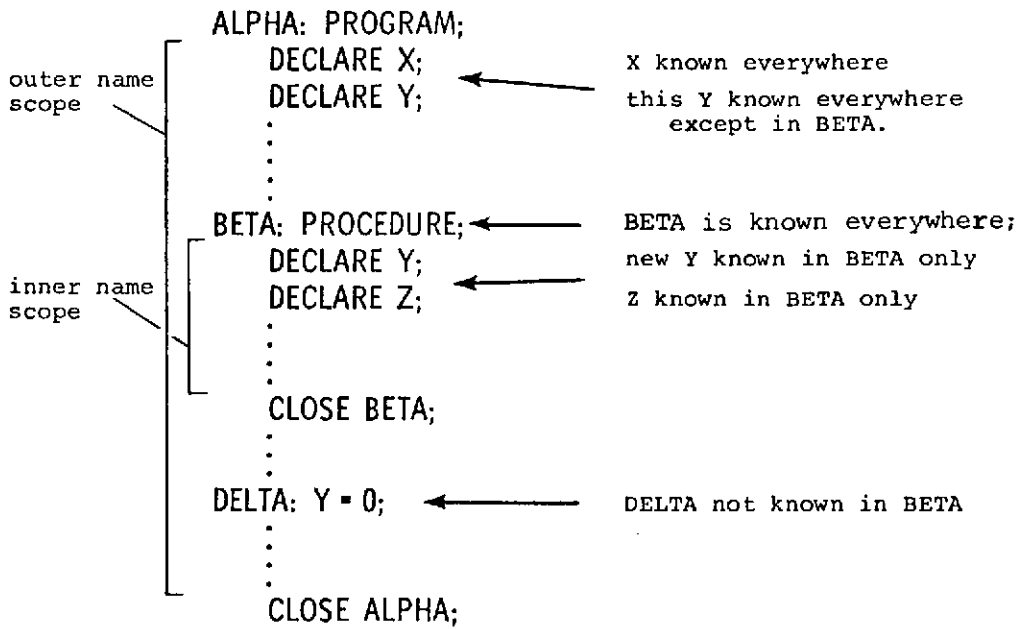
GENERAL RULES:

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:

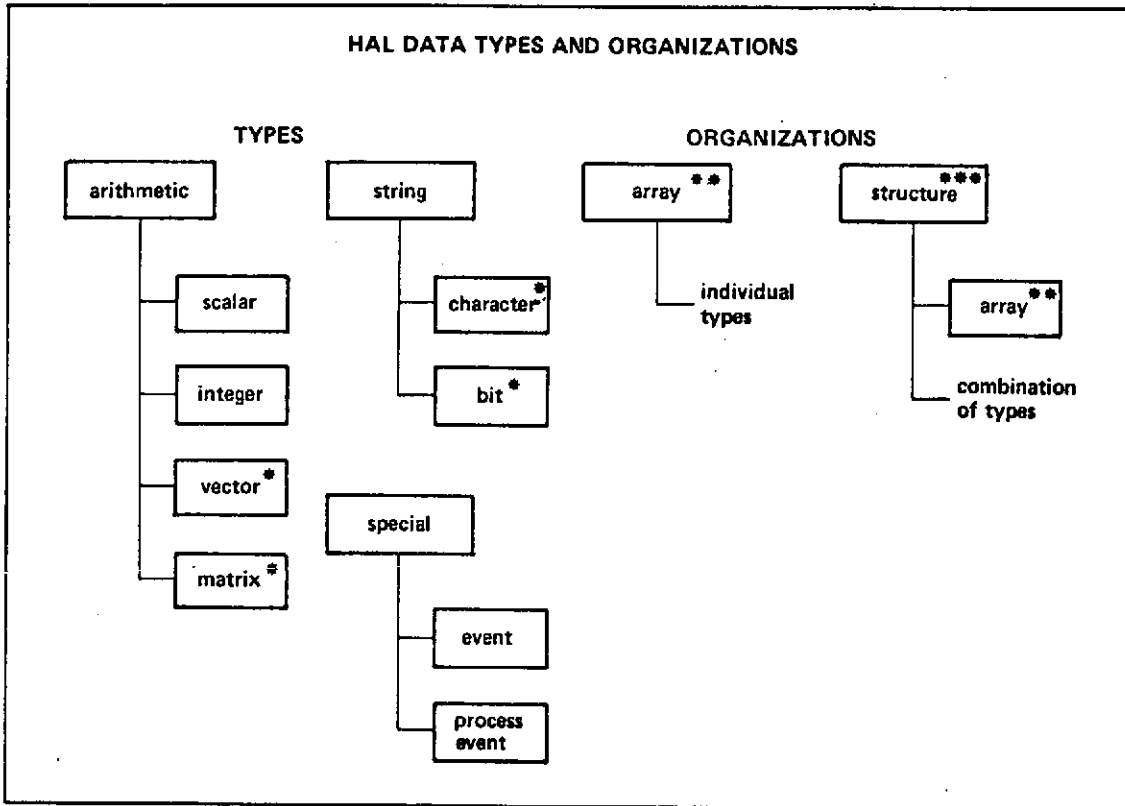
1. The name of a code block is taken to be defined in the name scope immediately enclosing the block. A PROCEDURE or FUNCTION label defined at the outermost level of compilation can be invoked from anywhere within the compilation.
2. The <label> of a statement is effectively unknown in blocks contained 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:



4. DATA AND OTHER DECLARATIONS

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

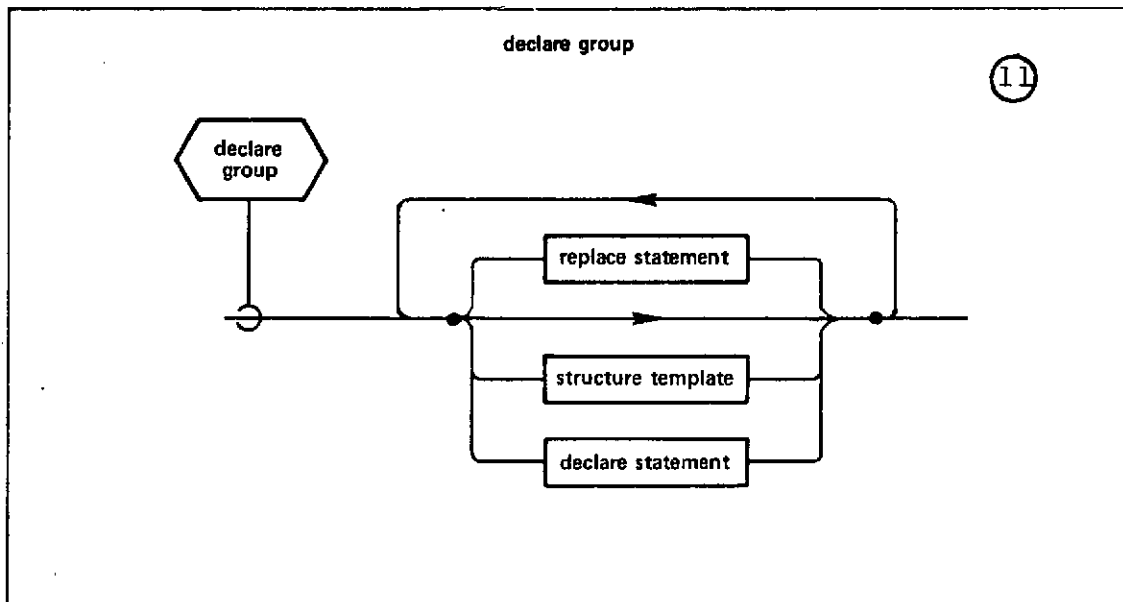


- * Component Subscripting (see Section 5.3.5) Allowed.
- ** Array Subscripting Allowed.
- *** Structure Subscripting Allowed.

4.1 The Declare Group.

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

SYNTAX:



SEMANTIC RULES:

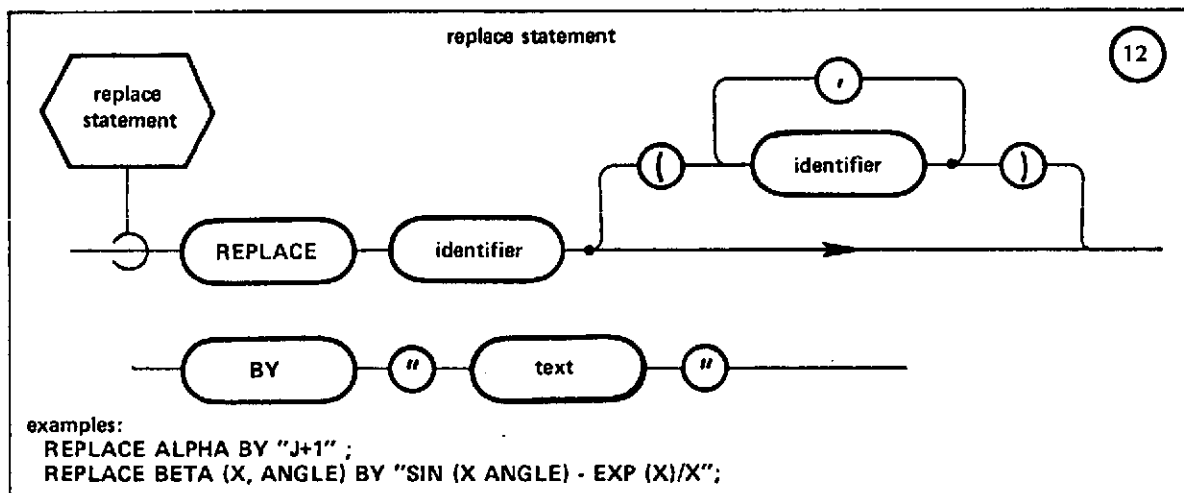
1. A <declare group> may simply be empty, or it may contain <replace statement>s, <structure template>s, and <declare statement>s. The form of each of these constructs is defined in this Section.
2. The "name scope" (see Section 3.8) of <identifier>s defined in a <declare group> is the code block containing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

4.2 The REPLACE Statement.

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

4.2.1 Form of REPLACE Statement

SYNTAX:



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

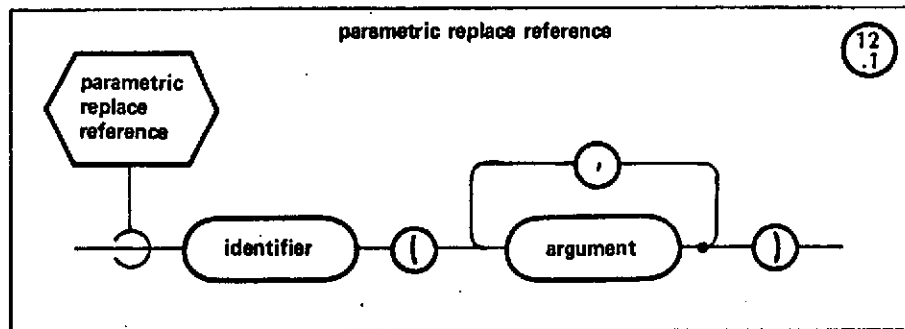
81

SEMANTIC RULES: Parametric Replacements

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

4.2.2 Referencing REPLACE Statements

SYNTAX:



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 ϕ signs (e.g. CBETA(A,B) ϕ) make 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.

- Blanks are significant in <argument>s. Only the commas used to separate <argument>s are excluded from the <text> values substituted into the macro definition.
- The <text> string comprising an <argument> may be empty. The value substituted in such a case is a null string.
- Within each <argument> there must be an even number of apostrophe characters ('). The effect of this rule is to require that each character literal used must be completely contained within a single <argument>.
- Within each <argument> there must be an even number of quotation mark characters ("). The effect of this rule is to require that the substitution of a nested REPLACE statement include the entire text of the replacement within a single <argument>.
- Within each <argument> there must be a balanced number of left and right parentheses: for each opening left parenthesis there must be a corresponding right parenthesis.
- Commas are not separators between <argument>s under the following circumstances:
 - within a character literal.
 - within REPLACE <text>.
 - nested within parentheses.

81

4.2.3 Identifier Generation

New identifiers may be generated by enclosing a reference to a simple REPLACE statement within ϕ signs. The effect is to make visible in the compiler listing, the catenation of the REPLACE \langle text \rangle with the characters surrounding the construct. For example, REPLACE ABLE BY "BAKER" then:

- 1) X = ϕ ABLE ϕ YZ
becomes X = BAKERYZ
- 2) CALL P_ ϕ ABLE ϕ (Q,R,S);
becomes CALL PBAKER(Q,R,S);

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

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 ϕ signs. The effect is the compile-time catenation of the corresponding macro argument with the characters surrounding the ϕ -enclosed parameter (a blank is considered as a character). For example:

81

```
REPLACE ABLE(X,Y) BY
  "P =  $\phi$ X $\phi$ QRS+Y;
  CALL SUB_ $\phi$ X $\phi$ ";
```

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

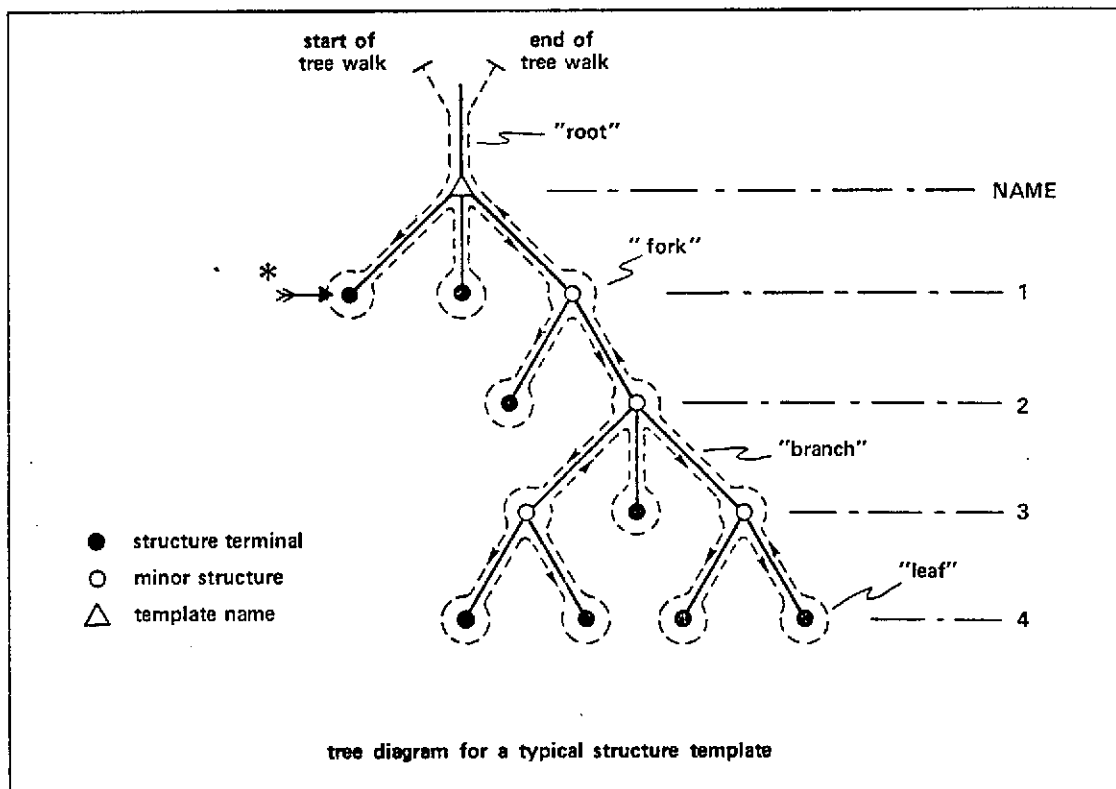
```
P = VQRS+A;
CALL SUB_V;
```

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

4.3 The Structure Template.

In HAL/S, a "structure" is a hierarchical organization of generally nonhomogeneous data items. Conceptually the form of the organization is a "tree", with a "root", "branches", and with the data as "leaves". The definition of the "tree organization" (the manner in which 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.

The following figure illustrates a typical tree organization.



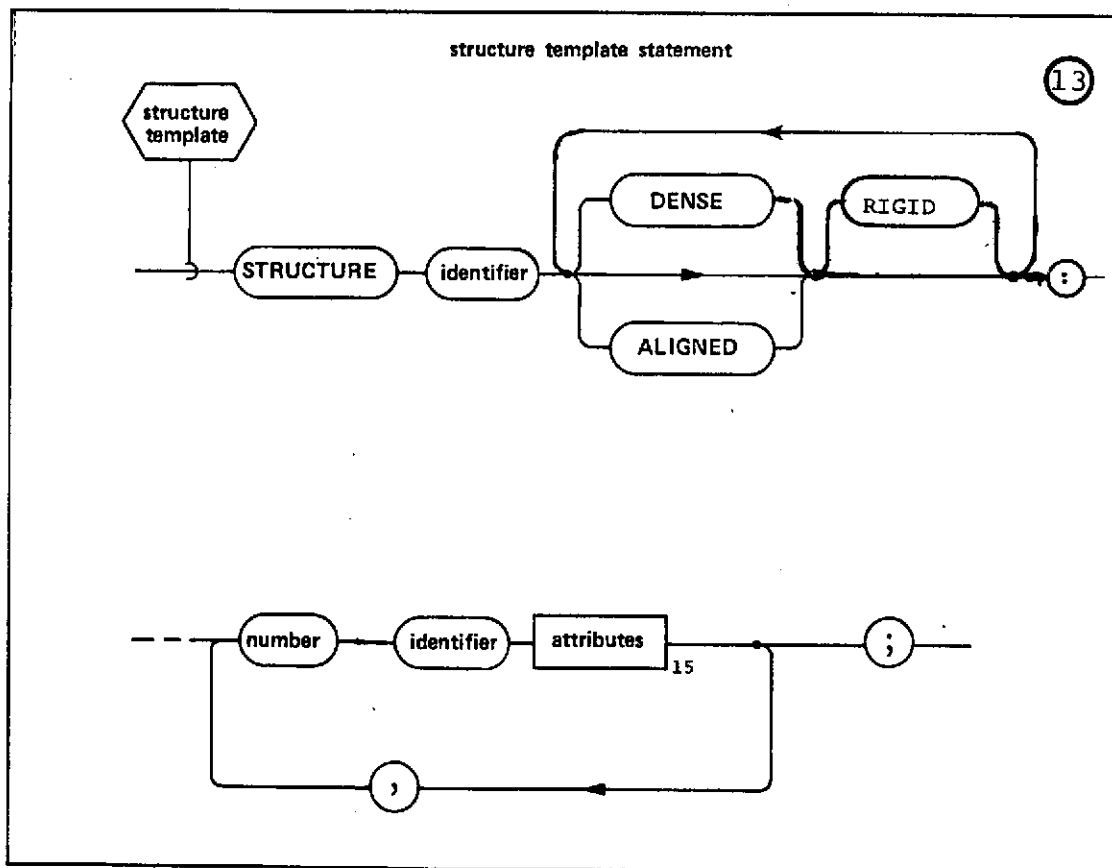
INTERPRETATIONS :

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 the diagram.
4. The tree organizations of two templates are considered to be equivalent for the purposes of various HAL/S statement contexts only if the tree forms are identical, and the type and attributes of all nodes in the tree agree. An implication of this rule becomes apparent: if two corresponding terminal nodes of otherwise equivalent structures reference different structure template names, then the structure templates containing these terminal nodes are not identical.

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

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 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.

E

NAME UNIQUENESS RULES:

1. <template names> may duplicate <identifiers> of any other kind within a given name scope, but may not duplicate other <template names>.
2. In a given name scope, if a <template name> is used exclusively in qualified structure declarations, duplications of the <identifiers> used for nodes may occur under the following circumstances:

- Any < identifier > used for a node in one template may duplicate an < identifier > used for a node in another template.
 - Any < identifier > used for a node in a given template may duplicate another < identifier > used for a different node in the same template, provided that a qualified reference can distinguish the two nodes.
3. In a given name scope, if a template is ever used for a non-qualified structure variable declaration, the duplications allowed under rule #2 within that template become illegal.

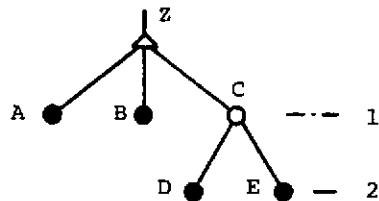
examples:

i) definition of a template Z

```

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

```

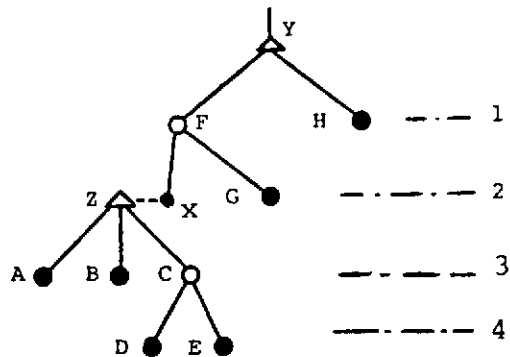


ii) definition of a template Y with Z nested within it

```

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

```

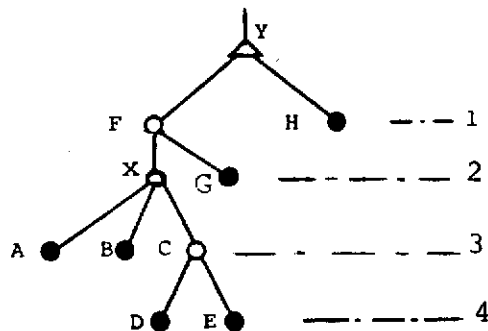


iii) equivalent form of template Y without nesting

```

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

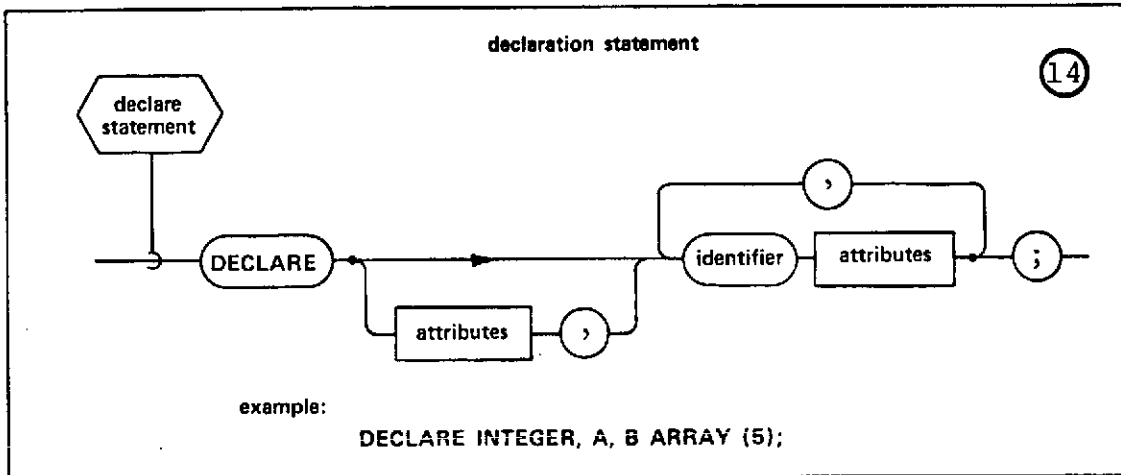
```



4.4 The DECLARE Statement.

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

SYNTAX:



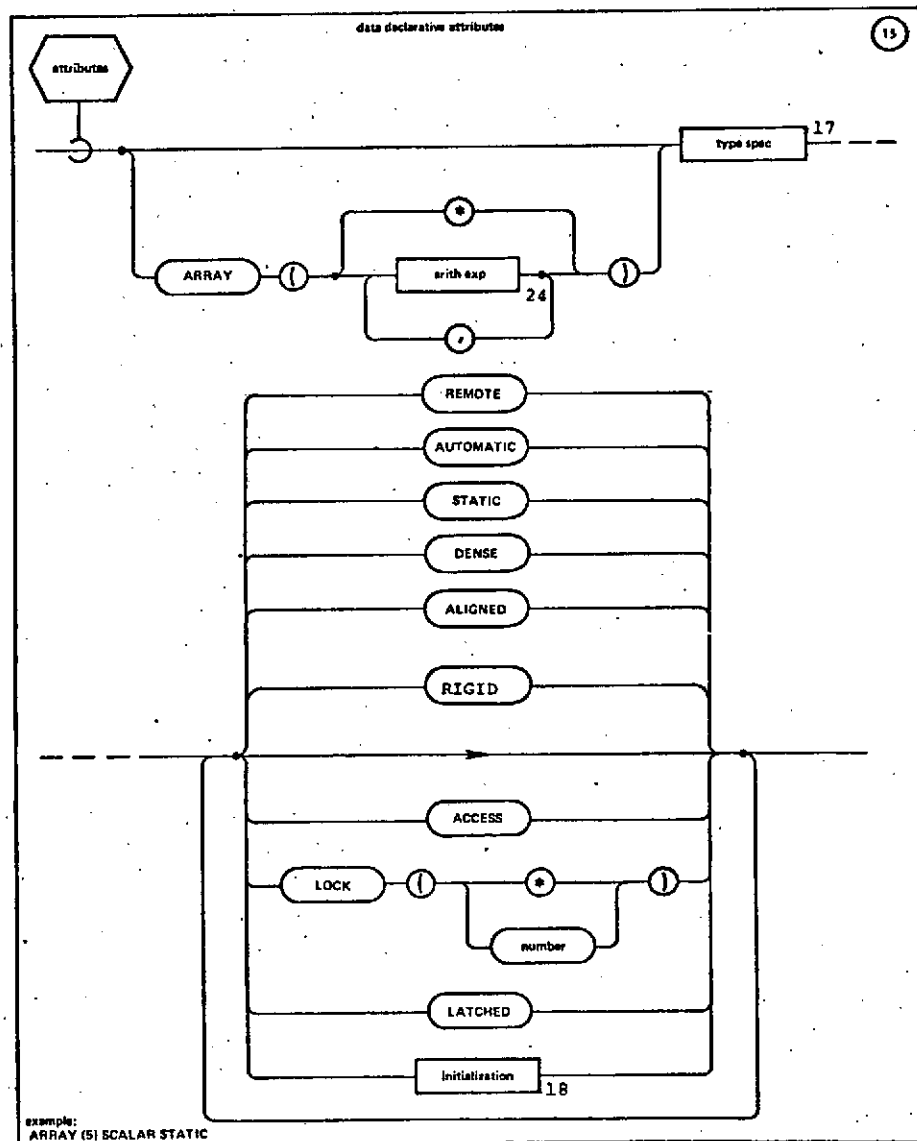
SEMANTIC RULES:

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 3.8). In any name scope all such < identifiers> must be unique.
4. There are two forms of <attributes>; data declarative, and label declarative. The form determines whether an <identifier> is defined as a data name or a label.

4.5 Data Declarative Attributes.

Data declarative attributes are used to define an <identifier> to be a data name or part of a structure template, and to describe its characteristics. If <attributes> 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:



90

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 time¹. 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:
 - STATIC/AUTOMATIC - the appearance of one of these keywords is mutually exclusive of the other. STATIC and AUTOMATIC refer to modes of initialization of an <identifier>, not to the allocation of its storage. The AUTOMATIC attribute causes an <identifier> with the <initialization> attribute to be initialized on every entry into the code block containing its declaration. The STATIC attribute causes such an <identifier> to be initialized only on the first entry into the code block. Thereafter its value on any exit from the code block is guaranteed to be preserved for the next entry into the block. STATIC data is not reinitialized whenever a program is re-entered (executed again). Values are preserved in this way even though a STATIC <identifier> has no <initialization>. Preservation of values is not guaranteed for AUTOMATIC <identifier>s. If neither keyword appears, then STATIC is assumed.

¹ See Appendix F.

- 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. 124
- ACCESS - this attribute causes implementation dependent managerial restrictions to be placed upon the usage of the <identifier> as a variable in assignment contexts. The manner of enforcement of the restrictions is implementation dependent.
- LOCK - this attribute causes use of the <identifier> to be restricted to the interior of UPDATE blocks, and to assign argument lists. <number> indicates the "lock group" of the <identifier> and lies between 1 and an implementation-dependent maximum. "*" indicates the set of all lock groups. The purpose of the attribute is described in Section 8.10.
- LATCHED - see Section 4.7.
- <initialization> - this attribute describes the manner in which the values of an <identifier> are to be initialized. It is described in Section 4.8.
- REMOTE - this attribute identifies data which is to be located in areas separate from normal data. Its implementation is machine dependent. Its purpose is to provide information to the compiler so that proper addressing to the data can be generated. Generally, this addressing requires longer and slower access methods. 124
- 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>. 90

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 LATCHED attribute only applies to event variables (see Section 4.7).
5. The REMOTE attribute is illegal for any <identifier> of EVENT type. It is also illegal if <identifier> is the input parameter of a PROCEDURE block.
6. The attributes DENSE, ALIGNED, and RIGID are illegal for major structures.
7. The <initialization> attribute may not be applied to formal parameters of procedures and functions.

RESTRICTIONS FOR STRUCTURE TERMINALS:

1. The asterisk form of array specification is not allowed.
2. The <identifier> may not be defined to be 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>.

90

EXAMPLE:

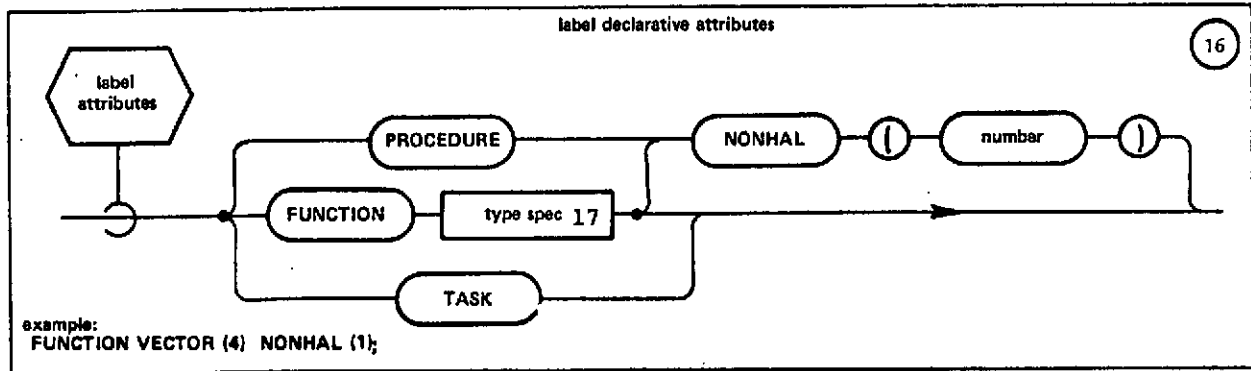
```
STRUCTURE Y:  
  1 A SCALAR,  
  1 B VECTOR(4),  
  1 D MATRIX(4,4);  
  
STRUCTURE Z RIGID;  
  1 F BIT(13),  
  1 G Y-STRUCTURE,  
  1 H CHARACTER(10);
```

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:

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

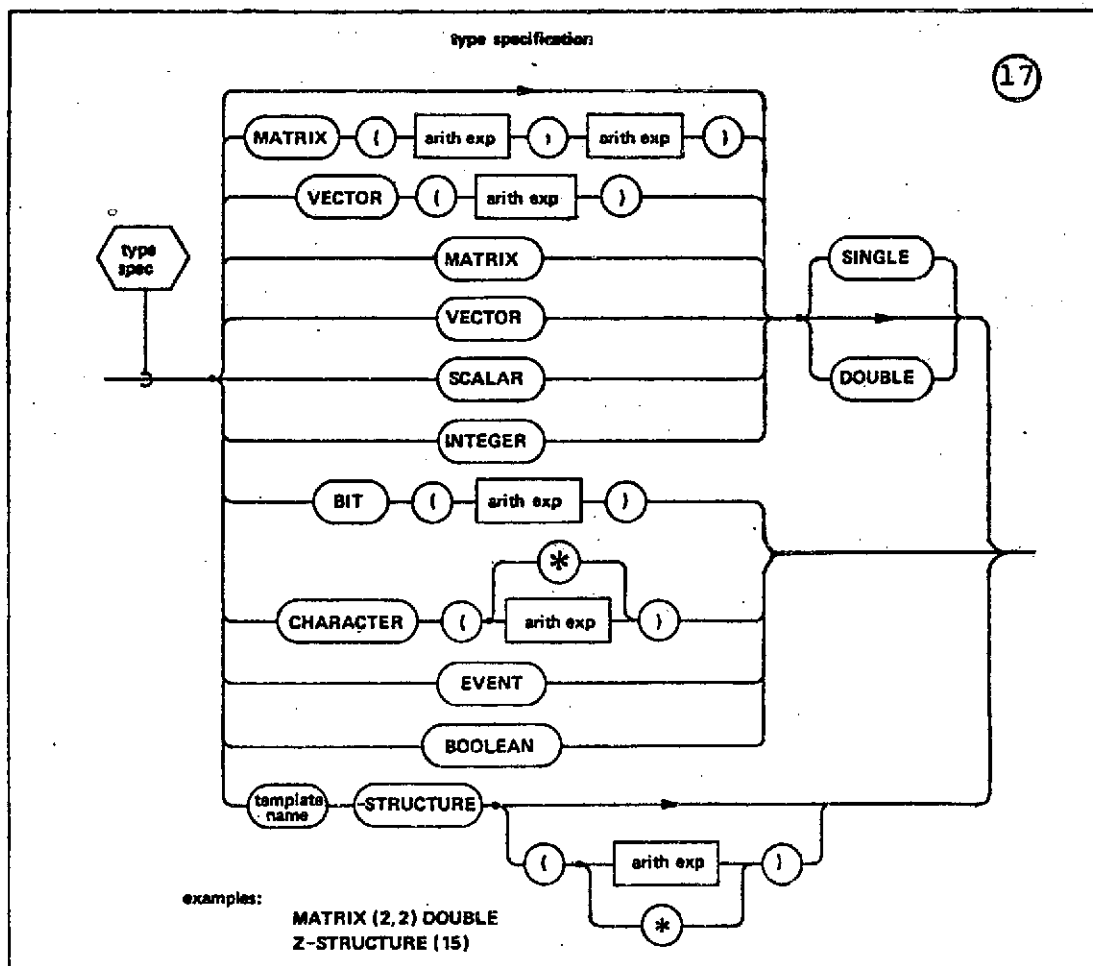
The type specification of the function declared is given by <type spec> (see Section 4.7). A function may be of any type except EVENT.

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

4.7 Type Specification.

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

SYNTAX:



GENERAL SEMANTIC RULES:

1. If <type spec> is empty (i.e. there is no specification present) then the interpretation is as follows:
 - If the <type spec> is that of a simple variable or structure terminal, then the implied type is SCALAR with SINGLE precision.

- 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 (see Appendix F.). Its value is rounded to the nearest integer.

RULES FOR INTEGER AND SCALAR TYPES:

- E |
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, BOOLEAN, AND EVENT TYPES:

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

RULES FOR STRUCTURE TYPE:

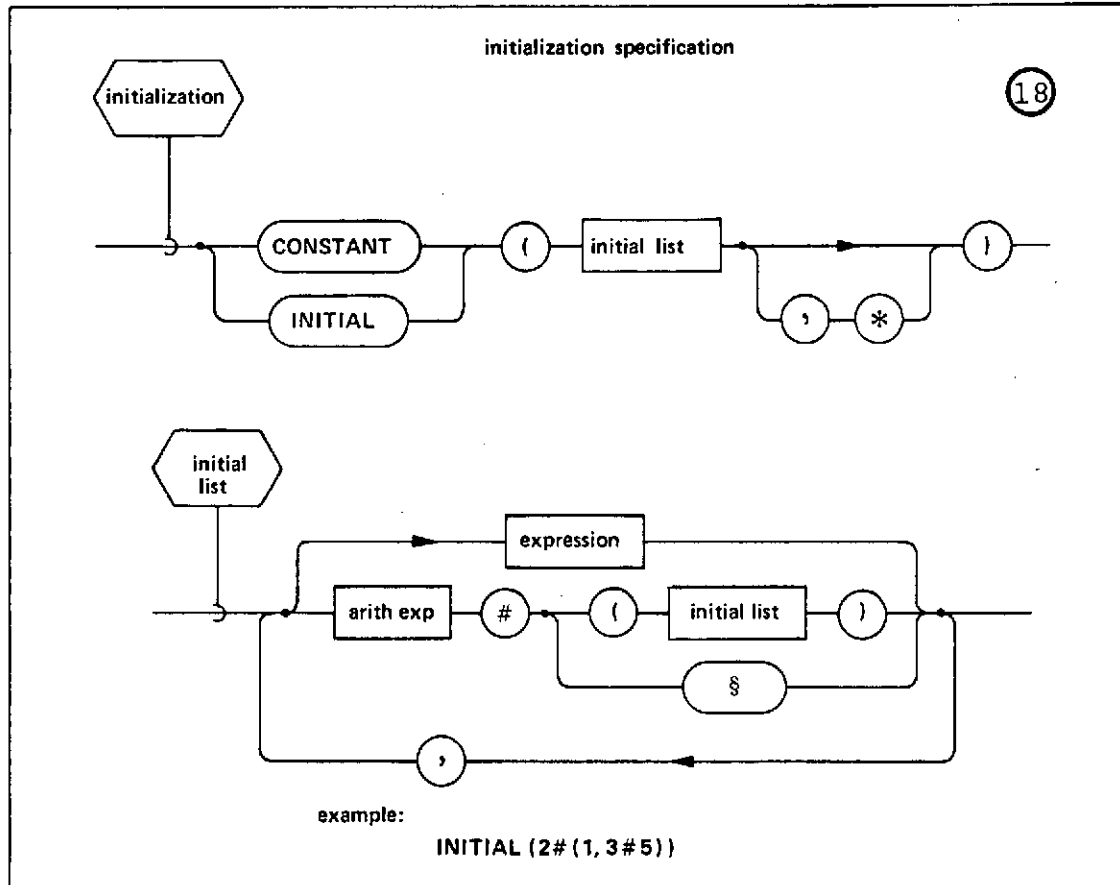
1. The condition for the <type spec> indicating a minor structure are described in General Semantic Rule 1.
2. The phrase <template name>-STRUCTURE defines an <identifier> to be a major structure whose tree organization is described by a previously defined template called <template name>.

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.

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:



GENERAL SEMANTIC RULES:

1. The <initialization> starts with the keyword INITIAL or CONSTANT. If it starts with CONSTANT, the value of the <identifier> initialized may never be changed. It is illegal for <identifier>s with CONSTANT <initialization> to appear in an assignment context.

2. The simplest form of an <initial list> is a sequence of one or more <expression>s computable at compile time. (See Appendix F).
3. A simple <initial list> of the form given in Rule 2. may 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 §. These items are:
 - a single literal;
 - a single unsubscripted variable name.
 - blank (i.e. the component(s) of the <identifier> should not be initialized).
5. The presence of an asterisk at the end of 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. <expression> must be an unarrayed integer or scalar expression computable at compile time (see Appendix F.). 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:
 - 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;
 - the number of values in the <initial list> is exactly equal to the number of components to be initialized;
 - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of components to be initialized, and partial initialization is indicated.
2. If the <identifier> has an array specification, then one of the following must hold:
 - the number of values in the <initial list> is exactly one, in which all the components of all the array elements of the vector or matrix are initialized to that value;
 - the number of values in the <initial list> is exactly equal to the number of components of the vector or matrix, in which case every array element takes on the same set of values.
 - the number of values in the <initial list> is equal to the total number of components in all array elements;

- the <initial list> ends with an asterisk, in which case the number of values must be less than the total number of components in all array elements, and partial initialization is indicated.
3. <expression> must be an unarrayed integer or scalar expression computable at compile time. Type conversion between integer and scalar is allowed where necessary.

RULES FOR BIT, BOOLEAN, EVENT AND CHARACTER TYPES:

1. If the <identifier> has no array specification, the <initial list> must contain exactly one value.
2. If the <identifier> has an array specification, then one of the following must hold:
 - the number of values in the <initial list> is exactly one, in which case all elements of the array are initialized to that value;
 - the number of values in the <initial list> is exactly equal to the number of array elements to be initialized;
 - the <initial list> ends with an asterisk, in which case the number of values must be less than the number of array elements to be initialized, and partial initialization is indicated.
3. If an <identifier> of bit, boolean, or event type is being initialized, <expression> must be an unarrayed <bit exp> computable at compile time (see Appendix F.). If an event <identifier> has the LATCHED attribute, then it may be initialized to the values TRUE or FALSE (or their equivalent). If it does not have the LATCHED attribute, it can not be initialized. (See Section 8.8). In the absence of <initialization> all events are implicitly initialized to FALSE.
4. If an <identifier> of character type is being initialized, <expression> must be an unarrayed <char exp> computable at compile time (see Appendix F.).

RULES FOR STRUCTURE TYPES:

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

5. DATA REFERENCING CONSIDERATIONS

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

5.1 Referencing Simple Variables.

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

5.2 Referencing Structures.

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

- If a structure is "unqualified", then any part of it, either minor structure or structure terminal, may be referenced by using the name of the part as it appears in the <structure template>. If a minor structure is referenced, the name becomes syntactically a <structure var name>. If a structure terminal is referenced, then syntactically the name becomes a <\$var name>, where \$ stands for any of the types arithmetic, bit, character, or event, as specified in its <attributes> in the template.
- If 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 <\$var name>, where \$ stands for any of the types arithmetic, bit, character, or event, as specified in its <attributes> in the template.

example:

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

} structure template

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

i) references to parts of structure A -

G I J

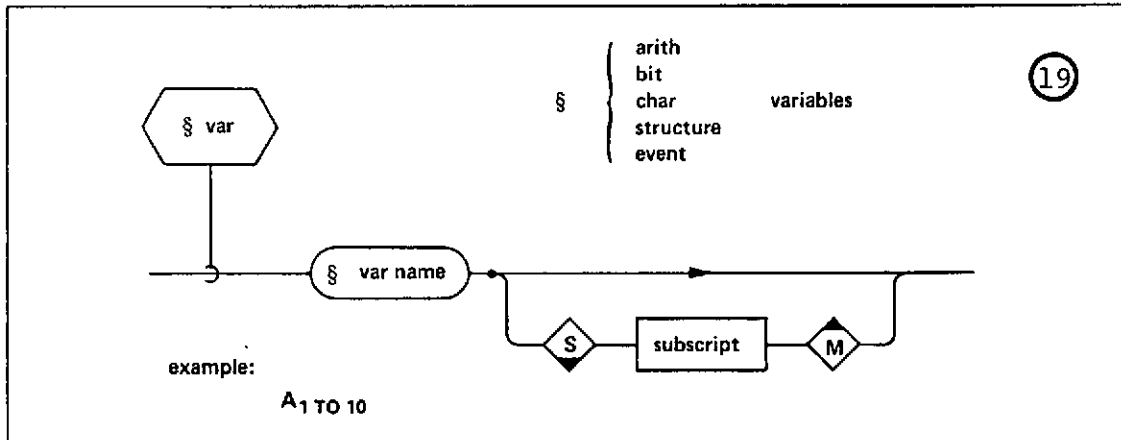
ii) references to corresponding parts of structure Z -

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

5.3 Subscripting.

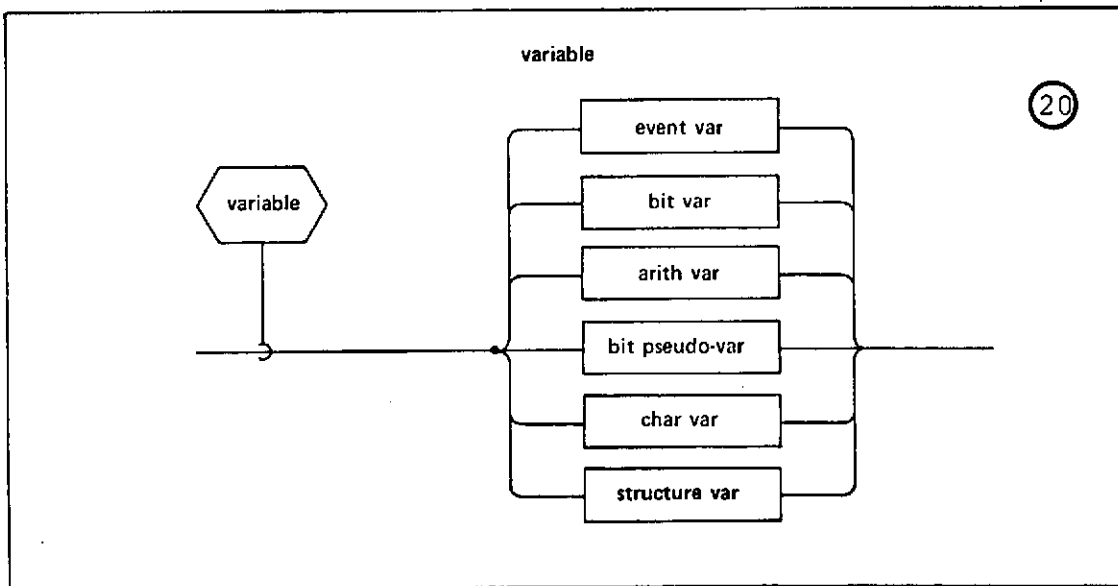
For the remainder of this Section, a data name with known <attributes> is denoted syntactically by <§var name>, where § stands for any of the types arithmetic, bit, character, event, or structure. It is convenient to introduce the syntactical term <§var> to denote any subscripted or unsubscripted <§var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

SYNTAX:



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

SYNTAX:



SEMANTIC RULES:

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

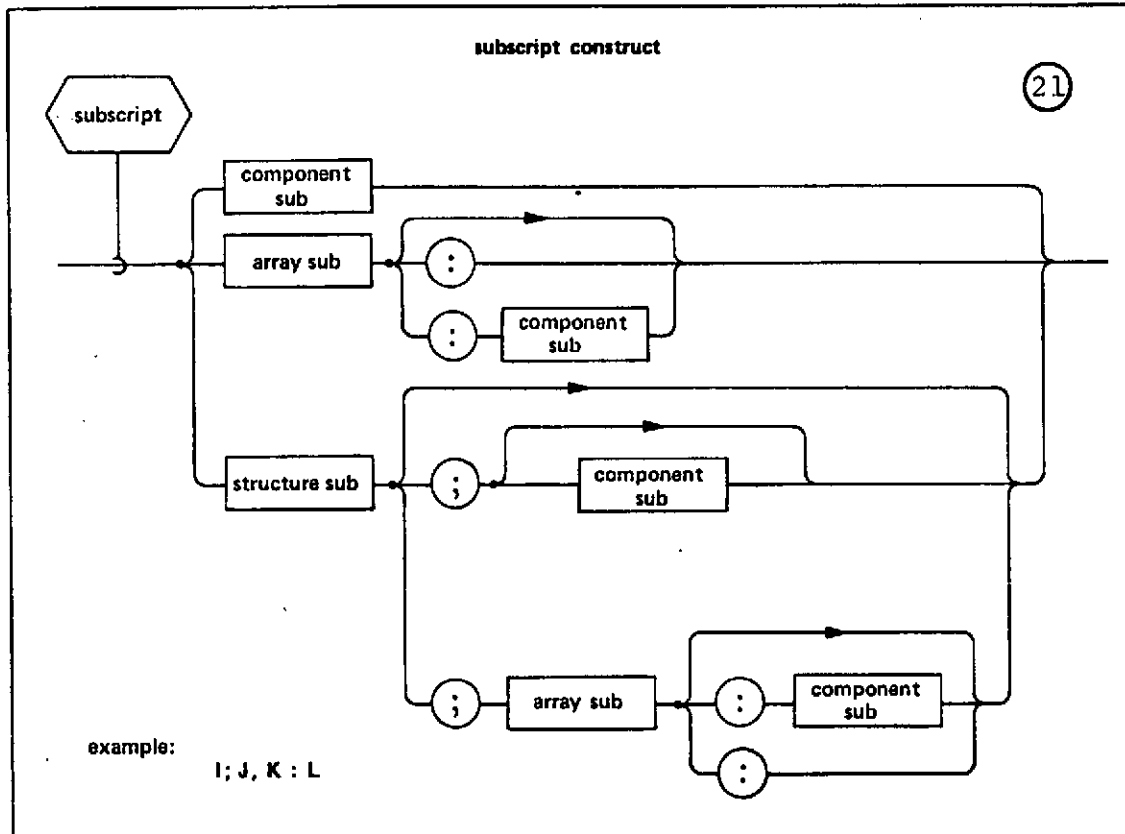
5.3.1 *Classes of Subscripting.*

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

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

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

SYNTAX:



SEMANTIC RULES:

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

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

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

Notes:

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

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

S
S;

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

examples:

i) $P_x:$
 \uparrow \leftarrow \langle array sub \rangle $\left|$ P is any arrayed simple variable

equivalent form -

P_x $\left|$ equivalent only if P is of integer, scalar, or event type

ii) Q_x
 \uparrow $\left\{ \begin{array}{l} \langle$ component sub $\rangle \left| Q \text{ is any simple variable of integer, scalar, or event type} \right. \\ \langle$ array sub $\rangle \left| \text{see example i)} \right. \\ \langle$ structure sub $\rangle \left| Q \text{ is any unarrayed structure terminal* of integer, scalar, or event type} \right. \end{array} \right.$

iii) $R_x;$
 \uparrow \leftarrow \langle structure sub \rangle $\left| R \text{ is any structure terminal*}$

equivalent forms -

R_x $\left|$ equivalent only if R is of unarrayed integer, scalar, or event type

iv) $S_{x;y;z}$
 \uparrow \leftarrow \langle component sub \rangle $\left| S \text{ is an arrayed structure terminal* of vector, matrix, bit, or character type} \right.$
 \uparrow \leftarrow \langle array sub \rangle
 \uparrow \leftarrow \langle structure sub \rangle

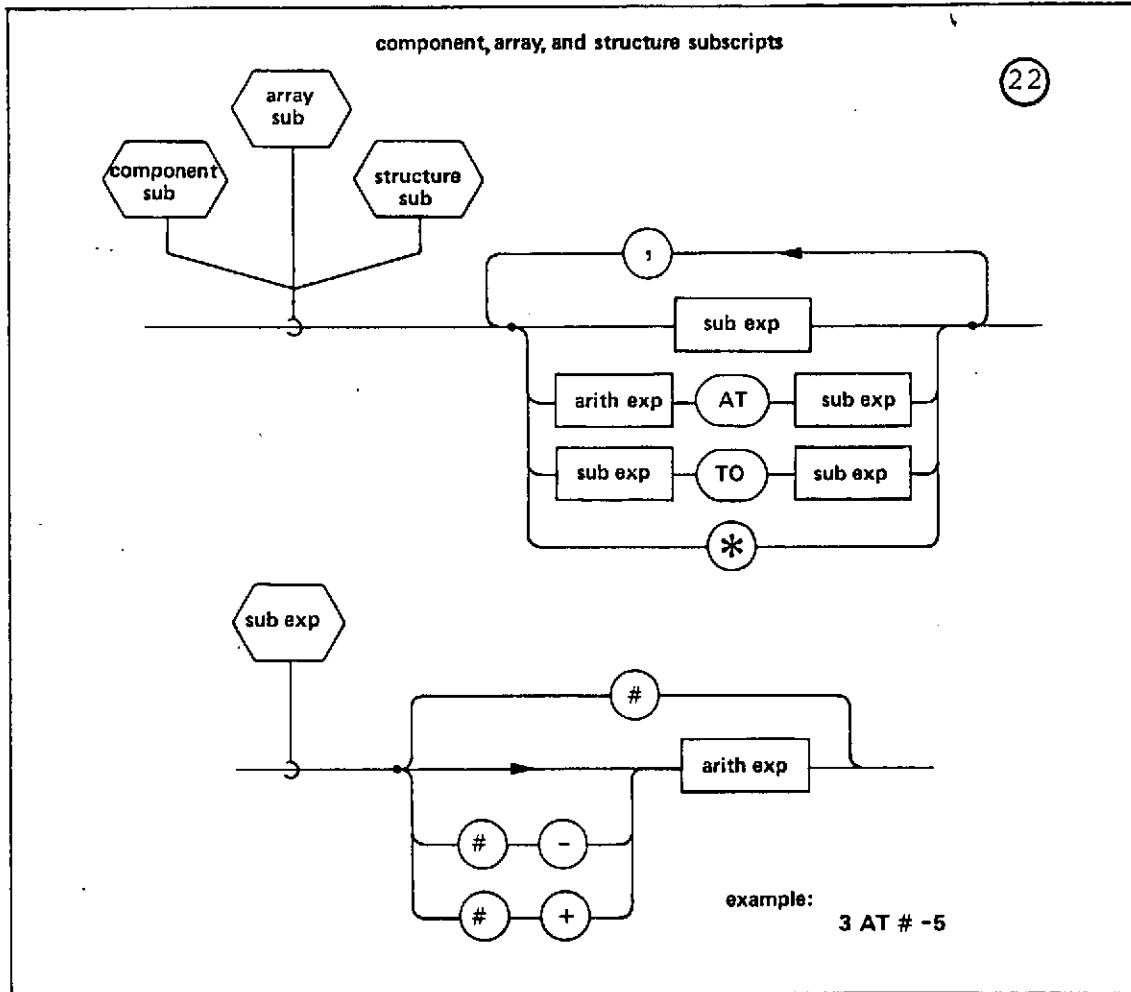
* of a structure with multiple copies

E

5.3.2 The General Form of Subscripting.

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

SYNTAX:



GENERAL SEMANTIC RULES:

1. A <structure sub>, <array sub>, or <component sub> consists of a series of "subscript expressions" separated by commas. Each subscript expression corresponds to a structure, array, or component dimension of the <\$var name> subscripted.

2. There are four forms of subscript expression:
 - the simple index;
 - the AT-partition;
 - the TO-partition;
 - the asterisk.
3. The simple index form is denoted in the diagram by a single sub exp . Its value specifies the index of a single component, array element, or structure copy to be selected from a dimension.
4. The AT-partition is denoted by the form <arith exp> AT <sub exp>. The value of <arith exp> is the width of the partition, and that of <sub exp> the starting index.
5. The TO-partition is denoted by the form <sub exp> TO <sub exp>. The two <sub exp> values are the first and last indices respectively of the partition.
6. The asterisk form, denoted in the diagram by *, specifies the selection of all components, elements, or copies from a dimension.
7. <sub exp> may take any of the forms shown. The value of # is taken to be the maximum index-value in the relevant dimension.
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.
 - in the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time (see Appendix F.).
 - in the form <sub exp> TO <sub exp>, the values of both <sub exp>s must be computable at compile time.

examples:

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

A_{20}	20 th copy of A
A_2 AT 10;	10 th and 11 th copies of A (semicolon optional)
C_1	C from 1 st copy of A
D_4 TO 6;	D from 4 th through 6 th copies of A (semicolon enforced)
Note: $D_{4:6}$ TO 6	components 4 through 6 of D from all copies of A

5.3.5 Component Subscripting.

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

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

RESTRICTIONS:

1. Errors result if any index value implied by a subscript expression lies outside the range 1 through N , where N is the size of the corresponding dimension in the type specification.
2. For bit, vector and matrix types, if the subscript expression is a TO- or AT-partition, the width of the partition must be computable at compile time. This is guaranteed by enforcing the following restrictions:
 - in the form <arith exp> AT <sub exp>, the value of <arith exp> must be computable at compile time.
 - in the form <sub exp> TO <sub exp>, the 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 $\langle \$var \text{ name} \rangle$ 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 $\langle \$var \text{ name} \rangle$ which is a structure terminal is said to be arrayed or to possess arrayness if either or both of the following hold:

- an array specification appears in its declaration in a structure template;
- the structure of which $\langle \$var \text{ name} \rangle$ is a terminal has multiple copies.

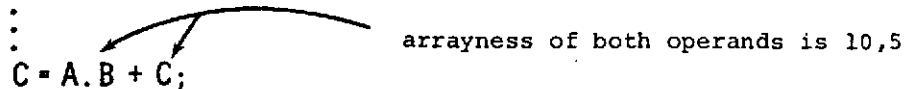
The number of dimensions of arrayness is the sum of the dimensions originating from each source.

Appending structure or array subscripting to a $\langle \$var \text{ name} \rangle$ may reduce the number and size of array dimensions of the resulting $\langle \$var \rangle$.

The arrayness of HAL/S expressions originates ultimately from the $\langle \$var \rangle$ s contained in them. It is a general rule that all arrayed $\langle \$var \rangle$ 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);  
:  
:  
:  
C = A.B + C;
```



arrayness of both operands is 10,5

5.4.1 Arrayness of Subscript Expressions

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

SEMANTIC RULES:

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

Example:

Given the declarations:

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

the following operands become:

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

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

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

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

① $I_1 = 2$

② $I_2 = 1$

③ $I_3 = 1$

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

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

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

However $MB_{*:I_1 \text{ TO } 2,*}$ is legal since array subscripting has been used on I to force arrayness matching.

If $MB \equiv \begin{pmatrix} [0.5 & 0.5] \\ [0.1 & 0.3] \\ [0.2 & 0.7] \\ [0.4 & 0.8] \end{pmatrix}$ ① $I_1 = 2$

then $MB_{*:I_1 \text{ TO } 2,*} \equiv \begin{pmatrix} MB_{1:2,*} \\ MB_{2:1,*} \end{pmatrix} \equiv \begin{pmatrix} [0.1 & 0.3] \\ [0.2 & 0.7] \end{pmatrix}$ ② $I_2 = 1$

5.5 The Natural Sequence of Data Elements.

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

The standard order in which this unraveling and reraveling takes place is called the "natural sequence". By applying the following rules in the order they are stated, the natural sequence of unraveling is obtained. By applying the rules in reverse order, and 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:  
  1 B,  
  2 C SCALAR,  
  2 D VECTOR(3),  
  1 E INTEGER;  
DECLARE A A-STRUCTURE(3);
```

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

RULES FOR OTHER OPERANDS:

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

example:

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

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

(standard HAL/S subscript notation used)

6. DATA MANIPULATION AND EXPRESSIONS

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

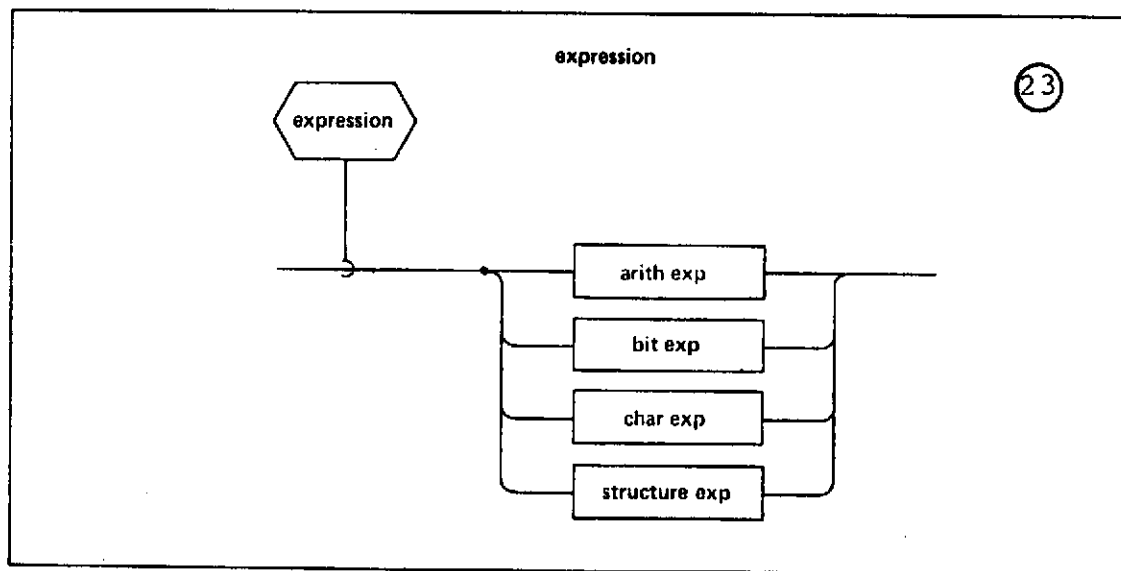
In HAL/S, expressions are divided into three major classes according to their usage.

- regular expressions appear in a very large number of contexts 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>.
- conditional expressions are used to express combinations of relationships between quantities, and are found in IF statements, and in WHILE and UNTIL phrases. They are denoted by <condition>.
- event expressions are used exclusively in real time programming statements.

6.1 Regular Expressions.

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

SYNTAX:

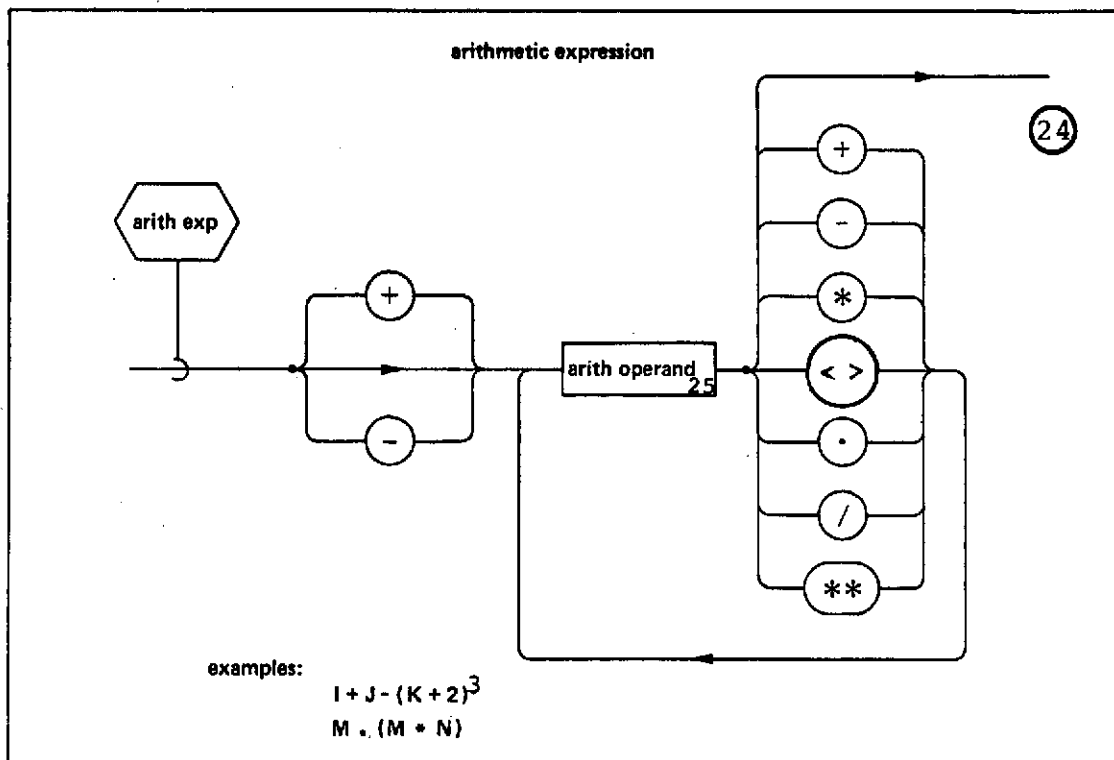


Descriptions of <arith exp>, <bit exp>, <char exp>, and <structure exp> are given in the following subsections.

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:



124

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.

124

- 124
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. The following table summarizes all possible forms, by indicating the result of each legal operation.

operands		infix operator						
left	right	+	-	<>	*	.	/	**
vector	vector	vector	vector	matrix ¹	vector ²	scalar ³		
vector	matrix			vector				
matrix	vector			vector				
vector	integer			vector ⁴			vector ⁵	
integer	vector			vector ⁴				
matrix	matrix	matrix	matrix	matrix				
matrix	integer			matrix ⁴			matrix ⁵	matrix ⁶
integer	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.

- 124
- 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 literal, in which case the result is integer.

5. Except as noted in Rule 4 (†), if only one operand in an operation is of integer type, it is converted to scalar type (see Appendix D).
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. The following table summarizes the precedence rules for arithmetic operators:

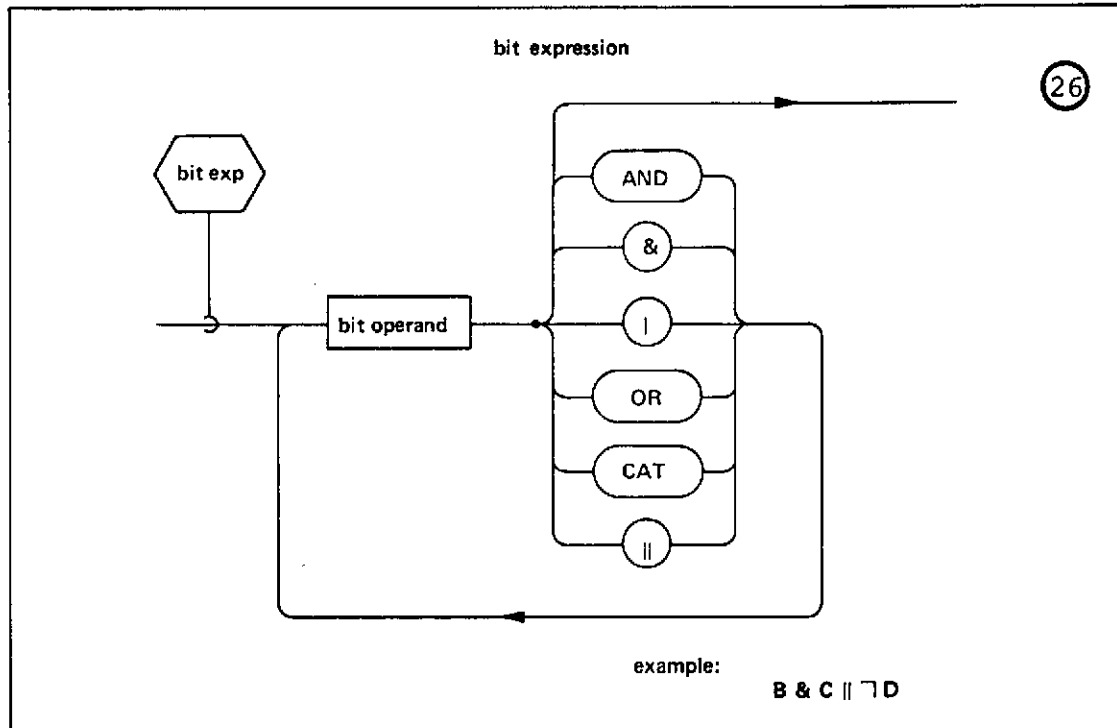
Operator	Precedence	Operation
	FIRST	
**	1	exponentiation
<>	2	multiplication
*	3	cross-product
.	4	dot-product
/	5	division
+	6	addition and unary plus
-	6	subtraction and unary minus
	LAST	

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

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:

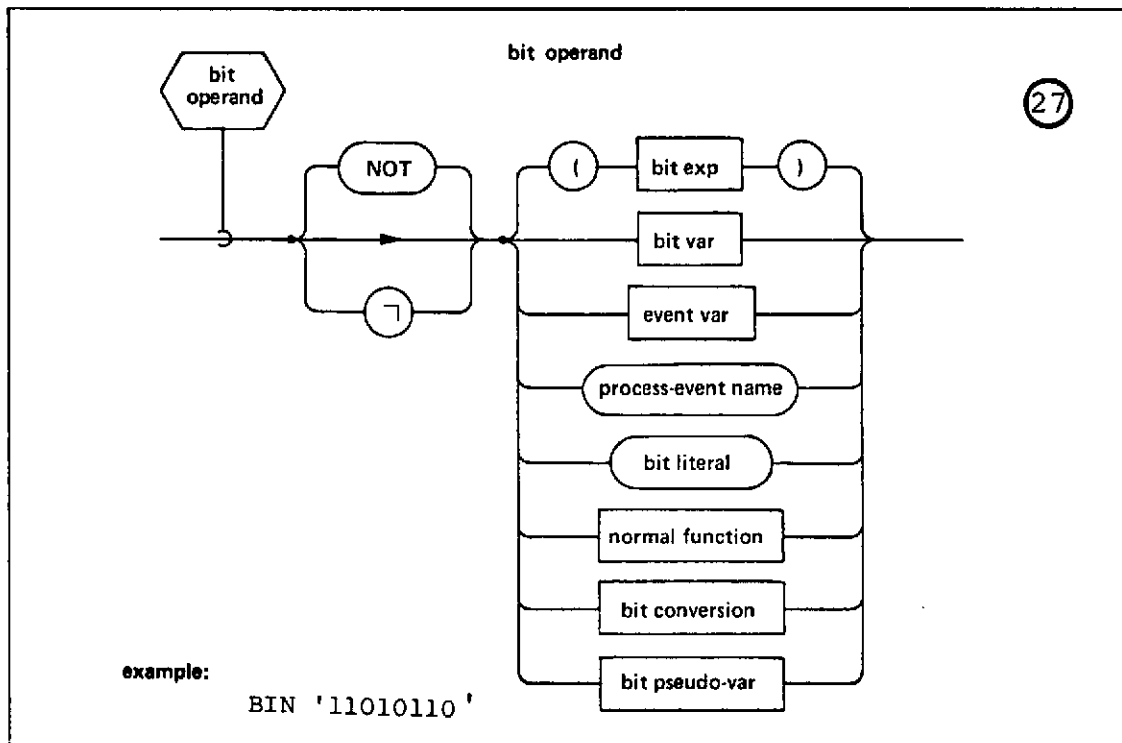
Operator	Precedence
	FIRST
CAT,	1
AND, &	2
OR,	3
	LAST

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

- The operator CAT (||) denotes catenation of <bit operand>s. The length of the result is the sum of the lengths of the operands.
- 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:



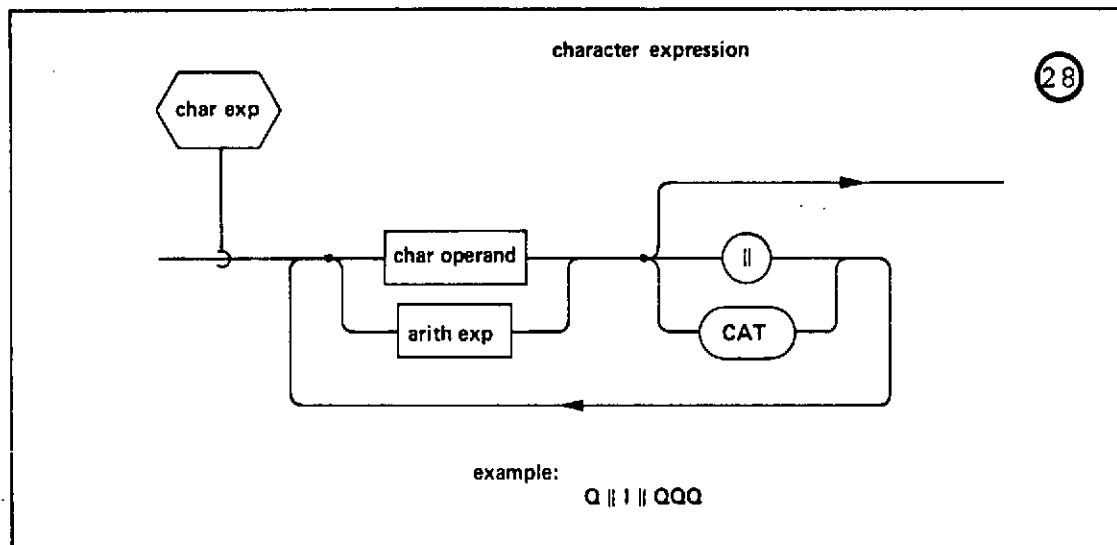
SEMANTIC RULES:

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

6.1.3 Character Expressions.

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

SYNTAX:

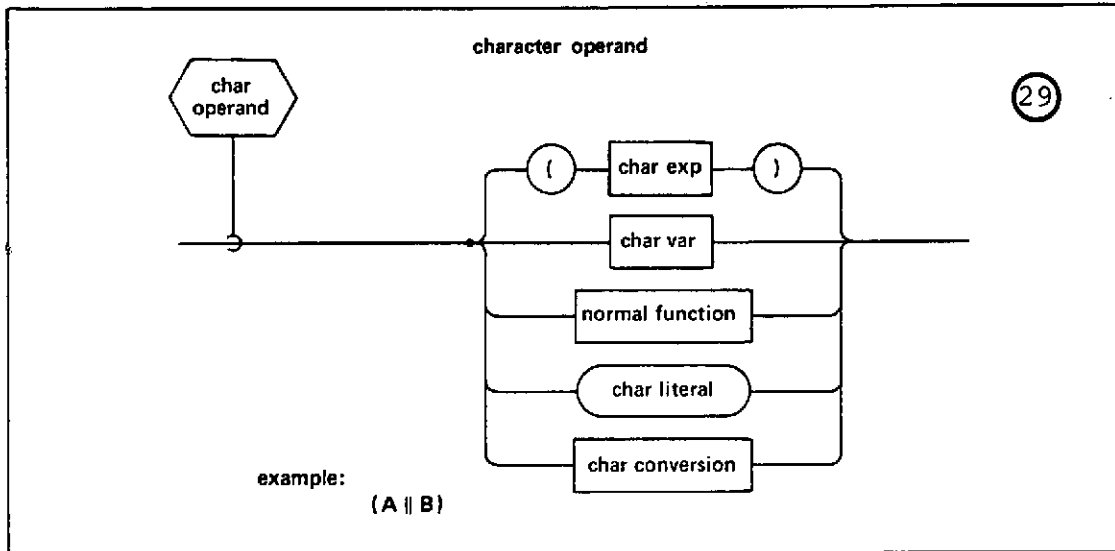


SEMANTIC RULES:

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

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

SYNTAX:



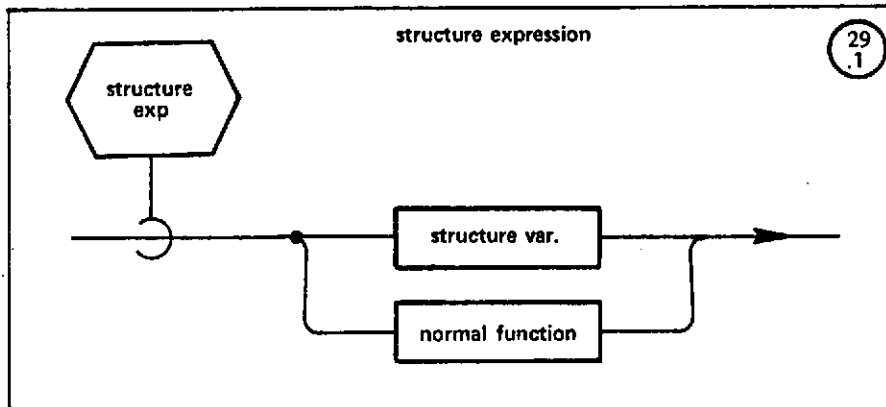
SEMANTIC RULES:

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

6.1.4 Structure Expressions

Since there are no manipulative expressions for 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.

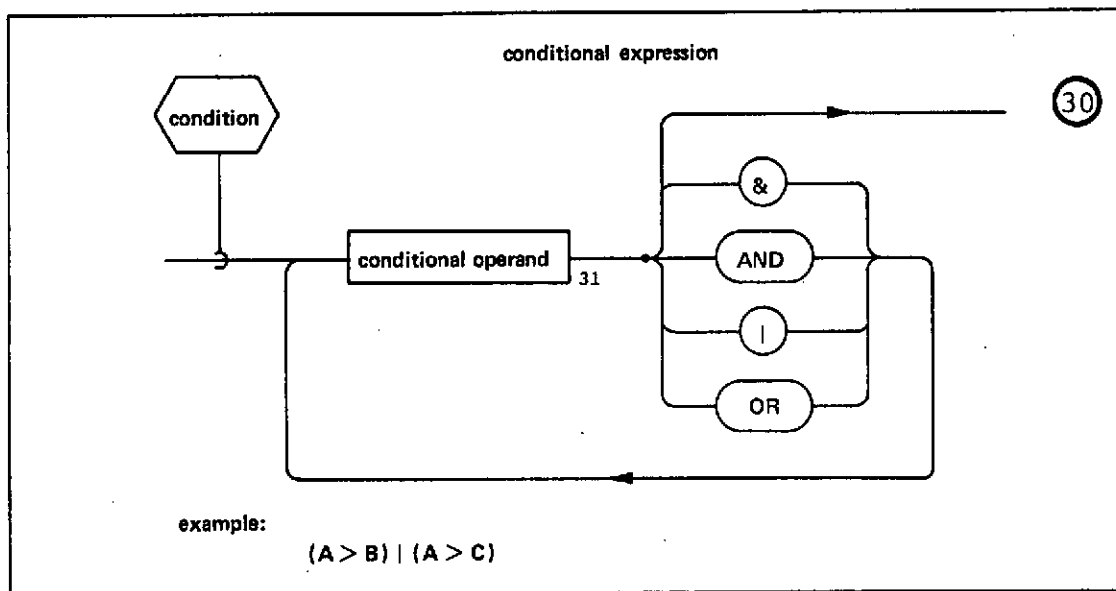
SEMANTIC RULES:

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

SYNTAX:



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:

124

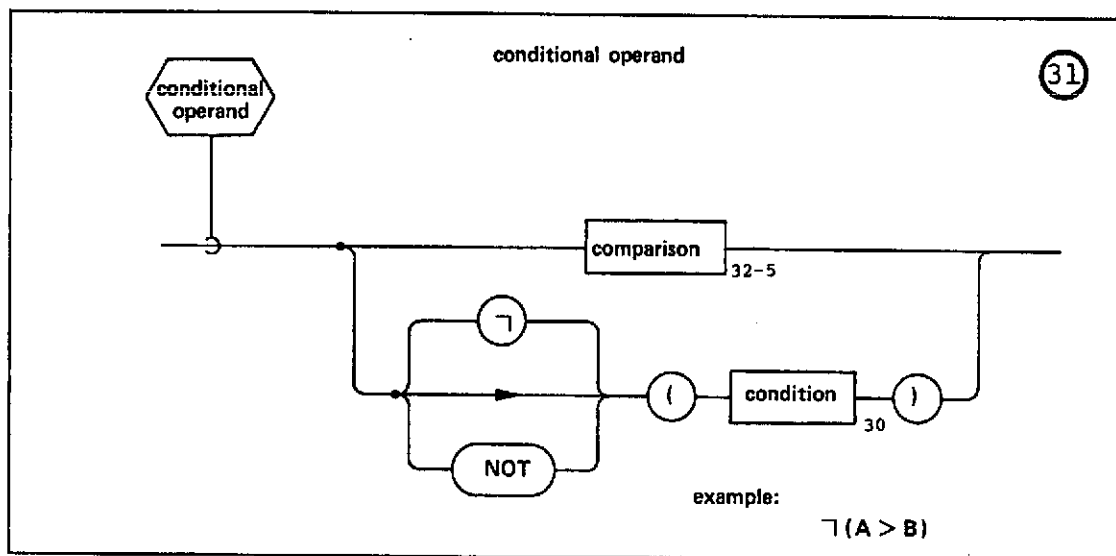
Operator	Precedence
	FIRST
AND, &	1
OR,	2
	LAST

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

- The operations AND (&) and OR (|) denote logical intersection and union respectively.

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

SYNTAX:



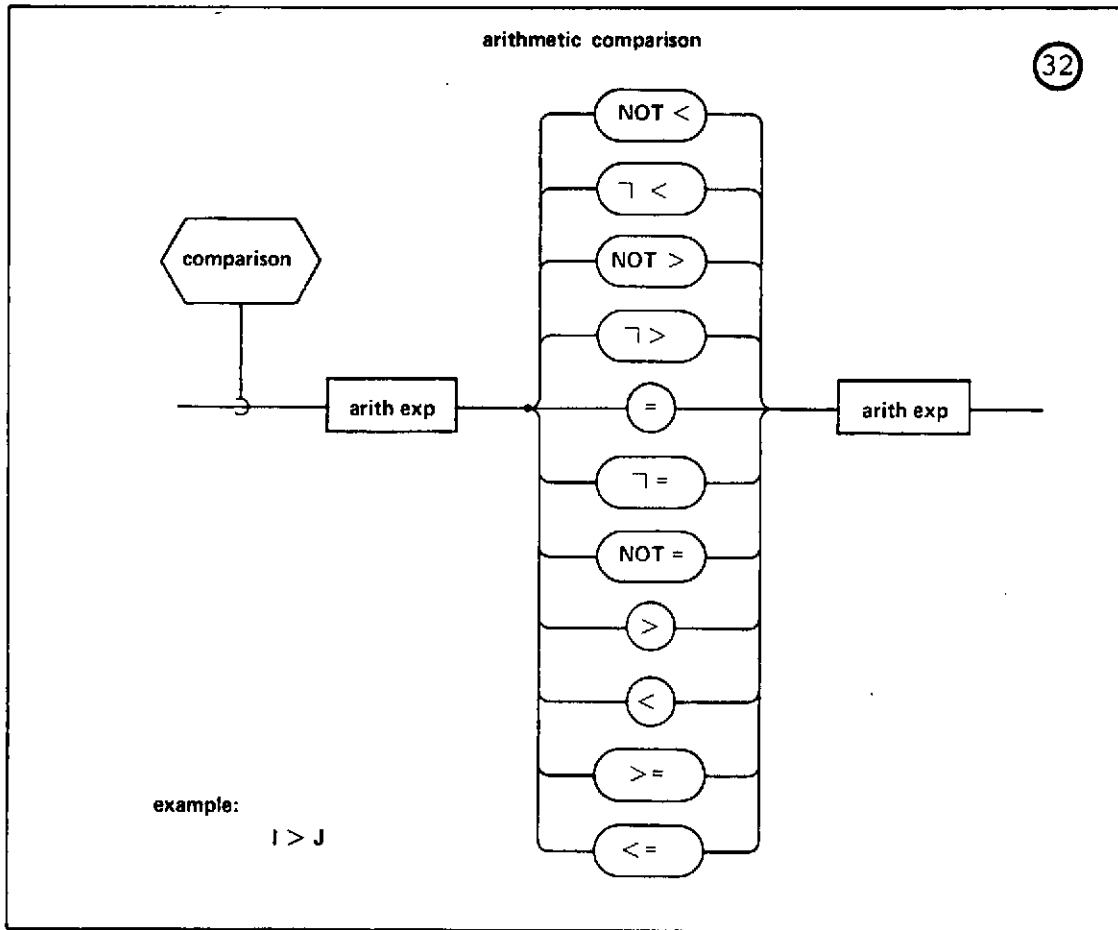
SEMANTIC RULES:

- A <conditional operand> is either a <comparison> or a parenthesized <condition>. The latter form may be preceded by the logical NOT (\neg) operator.
- 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 the following table indicate all legal forms.

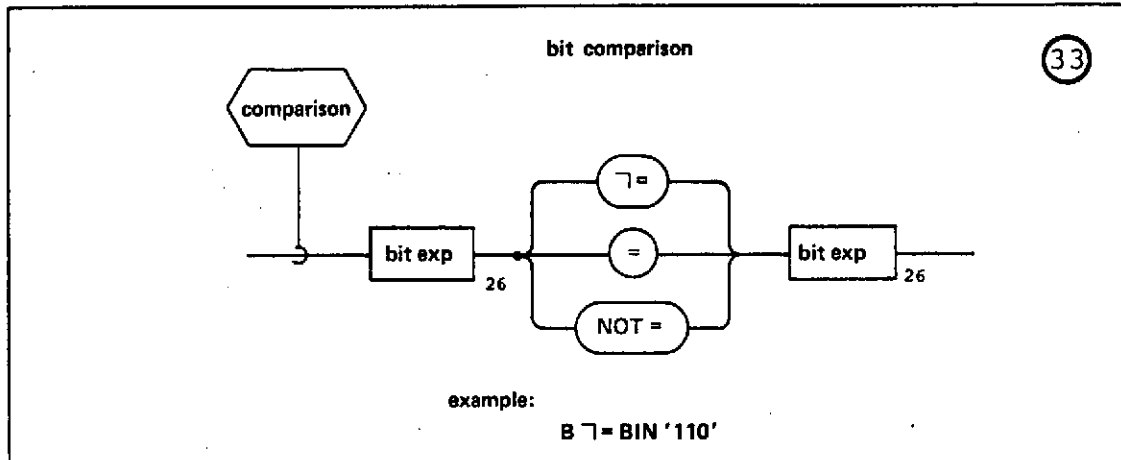
operands	operator						
	=	¬=	>	<	¬>	¬<	
					NOT>	NOT<	
					<=	>=	
vector	✓	✓					
matrix	✓	✓					
integer scalar	✓	✓	← no arrays →				

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

6.2.2 Bit Comparisons.

A bit comparison is a comparison between two bit expressions.

SYNTAX:



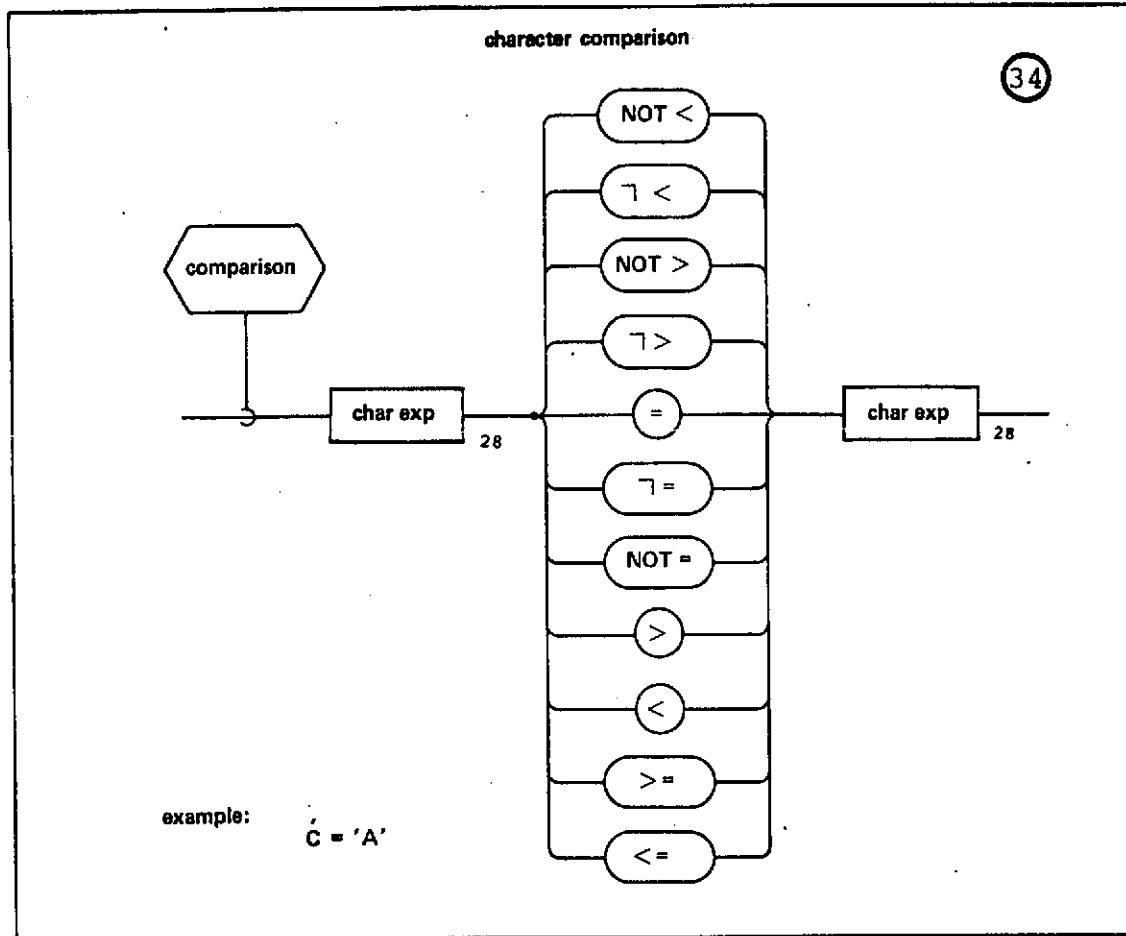
SEMANTIC RULES:

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

6.2.3 Character Comparisons.

A character comparison is a comparison between two character expressions.

SYNTAX:



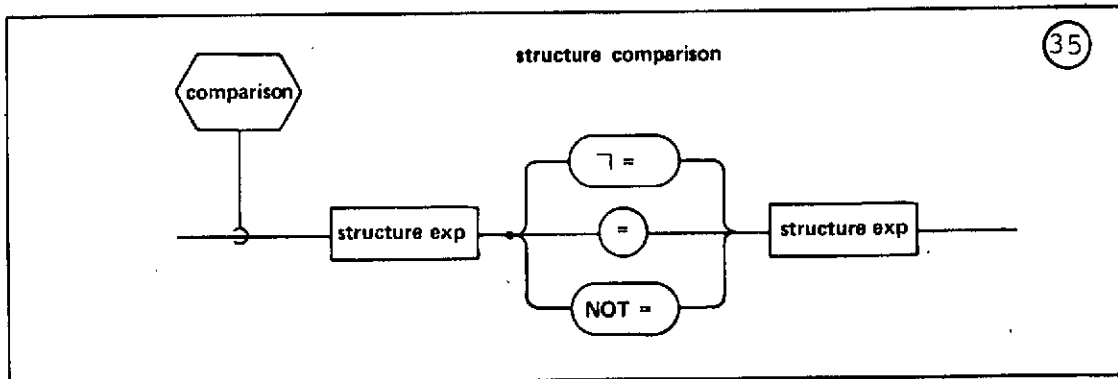
SEMANTIC RULES:

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 comparison 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:
 - if the <comparison> operator is =, the result is TRUE only if it is TRUE for all copies.
 - if the <comparison> operator is ≠ (NOT=), the result is TRUE if it is TRUE for at least one pair of corresponding copies.

6.2.5 Comparisons between Arrayed Operands.

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

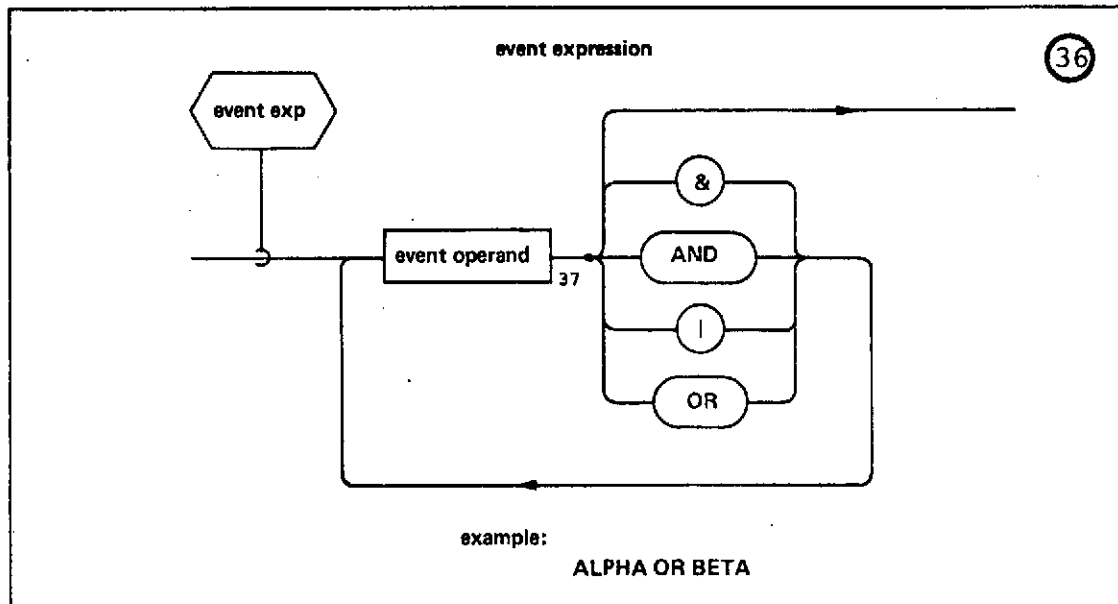
SEMANTIC RULES:

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 \neq (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:

E

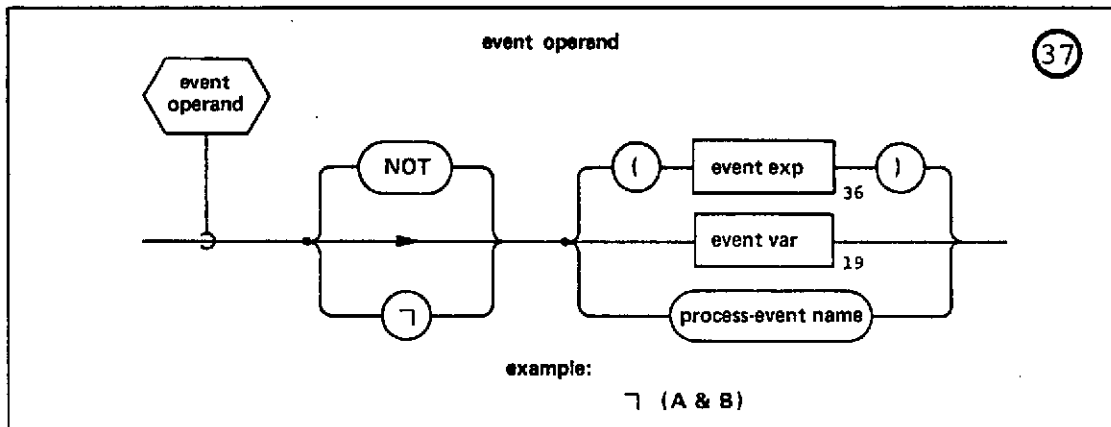
Operator	Precedence
	FIRST
AND, &	1
OR,	2
	LAST

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

- 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:

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

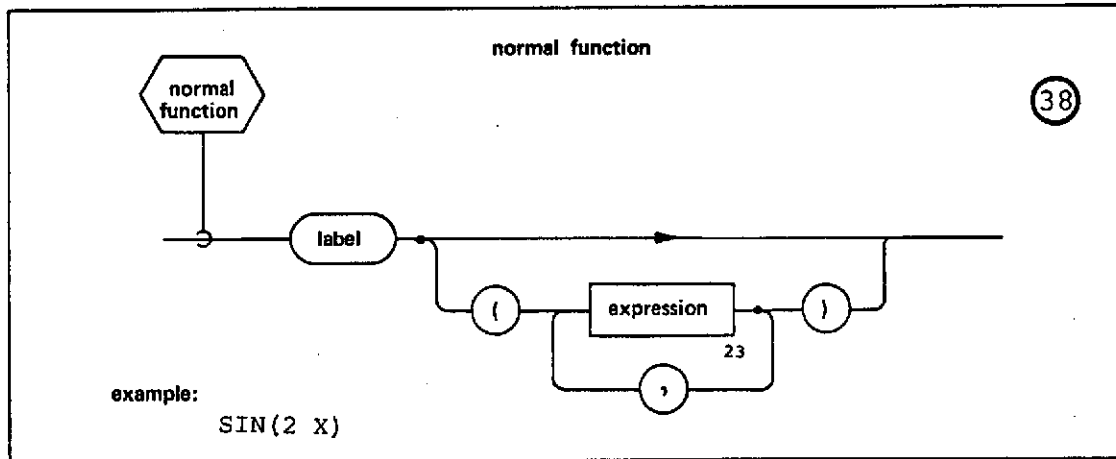
6.4 Normal Functions.

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

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

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

SYNTAX:



SEMANTIC RULES:

1. <label> invokes execution of a function with name <label>.
2. If <label> is a reserved word which is a built-in function name then that built-in function is invoked. A list of built-in function names is given in Appendix C.
3. If a <function block> with name <label> appears in such a name scope that <label> is known to the invocation, then that block is invoked.
4. If no such <function block> exists, then the <function block> is assumed to be external to the <compilation> containing the invocation. A <function template> for that <function block> must therefore be present in the <compilation> (see Section 3.6).

- 124
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"¹.
 7. Each input argument of a <normal function> must match the corresponding input parameter of the function definition² exactly in type, dimension, structure function, and tree organization, as applicable, except for the following relaxations:
 - precisions need not match, precision conversions are allowed;
 - the lengths of bit arguments need not match;
 - the lengths of character arguments need not match;
 - implicit integer to scalar and scalar to integer conversions are allowed;
 - 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.

1 See Section 7.4.

2 the parameter specifications for built-in functions is part of the formal definition given in Appendix C.

example:

```
DECLARE X ARRAY(4) SCALAR;  
.  
.  
[X] = SIN(X);  
.  
ADD: FUNCTION (P) SCALAR;  
DECLARE P ARRAY(4) SCALAR;  
.  
.  
RETURN P1+P2+P3;  
CLOSE ADD;  
.  
.  
[X] = [X] + ADD([X]);  
.  
.
```

SIN evaluated once
for each element of 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.

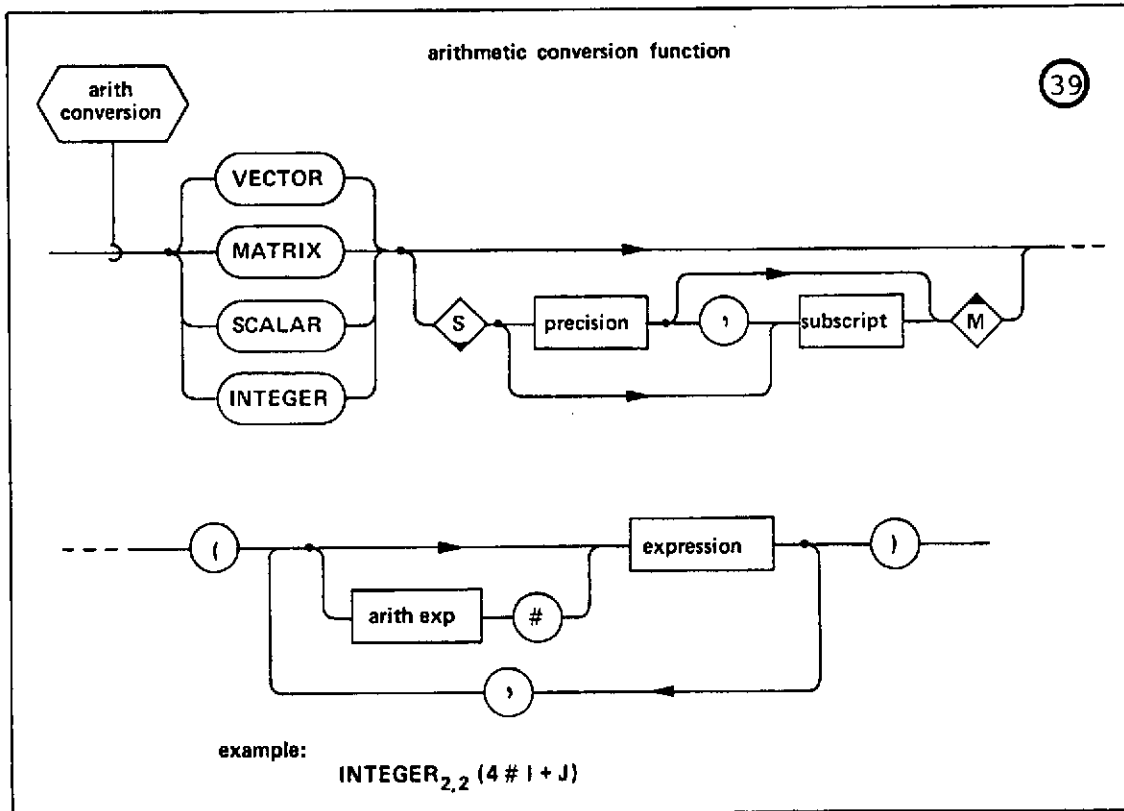
6.5 Explicit Type Conversions.

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

6.5.1 Arithmetic Conversion Functions.

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

SYNTAX:



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 6.6), and by a <subscript> specifying its dimensions.

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

SEMANTIC RULES (INTEGER and SCALAR);

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

³ see Section 5.5.

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 a <subscript> VECTOR produces a single 3-vector result; MATRIX produces a single 3-by-3 matrix result. The number of data elements implied by the argument(s) must therefore be equal to 3 and 9 respectively.
2. VECTOR and MATRIX cannot produce arrays of vectors and matrices. Consequently, <subscript> may only indicate terminal subscripting.
3. In VECTOR the <subscript> must be an <arith exp>. <arith exp> is an unarrayed integer or scalar expression computable at compile time (see Appendix F). Its value is rounded to the nearest integer, and must be greater than one. It denotes the length of the vector produced by the conversion. It must therefore match the total number of data elements implied by the argument(s) of the conversion.
4. In MATRIX the form of the <subscript> must be

<arith exp>,<arith exp>

Each <arith exp> is an unarrayed integer or scalar expression computable at compile time. Values are rounded to the nearest integer, and must be greater than one. They denote the row and column dimensions respectively of the matrix produced by the conversion. Their product must therefore match the total number of data elements implied by the argument(s) of the conversion.

5. VECTOR and MATRIX may have arguments of integer, scalar, vector, and matrix 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

```

```

INTEGER2,6(2#[X])    result is 2,6 array of integers*

```

```

MATRIX(3#V)            result is 3 by 3 matrix, each row
                        being equal to V

```

```

VECTOR6([X])         vector of length 6

```

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

* For example:

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

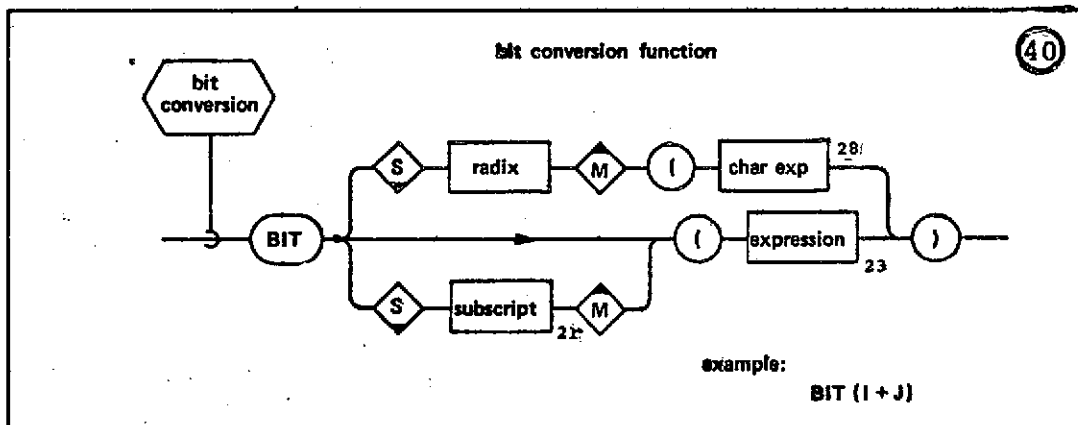
- Argument 2#[X] is "first unraveled", i.e.
 $[\underline{1\ 2\ 3\ 4\ 5\ 6}, \underline{1\ 2\ 3\ 4\ 5\ 6}]$
- Linear string is then "reraveled" into 2x6 array:

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

6.5.2 The Bit Conversion Function.

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

SYNTAX:



GENERAL SEMANTIC RULES:

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

SEMANTIC RULES (without <radix>):

1. Conversions of the argument proceed according to the standard conversion rules given in Appendix D. The resulting bit string is of maximum length for the implementation and the significant data is right justified within the word.
2. <subscript> represents component subscripting upon the results of the conversion. <subscript> has the same semantic meaning and restrictions in the current context as it does in the subscripting of bit <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>. <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
```

```
BIT1 TO 16([X]) same as above except that only bits  
1 through 16 of each array element  
are taken
```

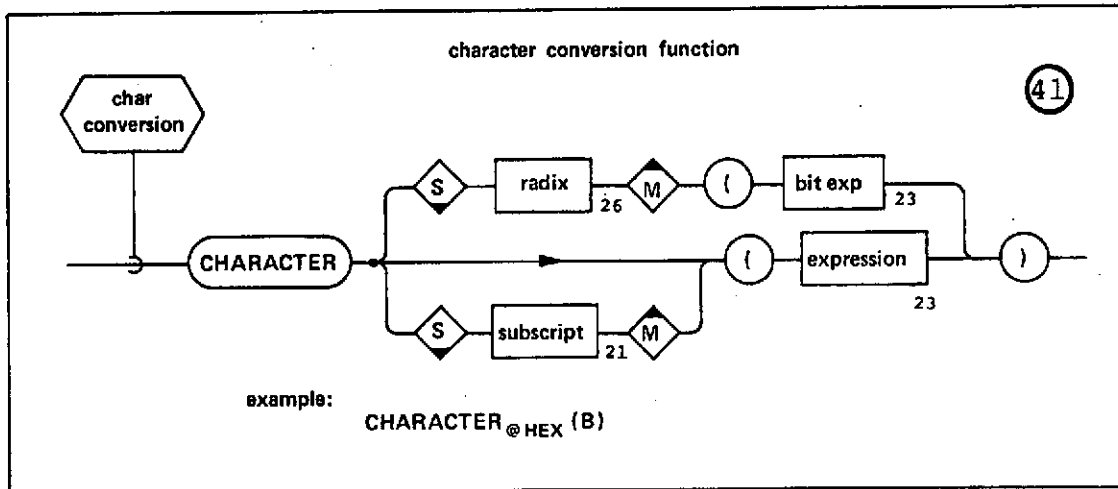
```
BIT@HEX('FACE') result is bit pattern of hexadecimal  
digits represented by argument
```

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

6.5.3 The Character Conversion Function.

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

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

SEMANTIC RULES (with <radix>):

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

@HEX	(hexadecimal)
@DEC	(decimal)
@OCT	(octal)
@BIN	(binary)
2. The value of <bit exp> is converted to the representation indicated by the <radix>, left padding the value with binary zeroes as required. The result is a character string consisting of the digits of the representation.

examples:

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

```
⋮
```

```
CHARACTER([X])
```

result is a 2,3 array of character strings

```
CHARACTER2([X])
```

same as above except that only the second character of each array element is taken

```
CHARACTER@DEC(BIN '101101')
```

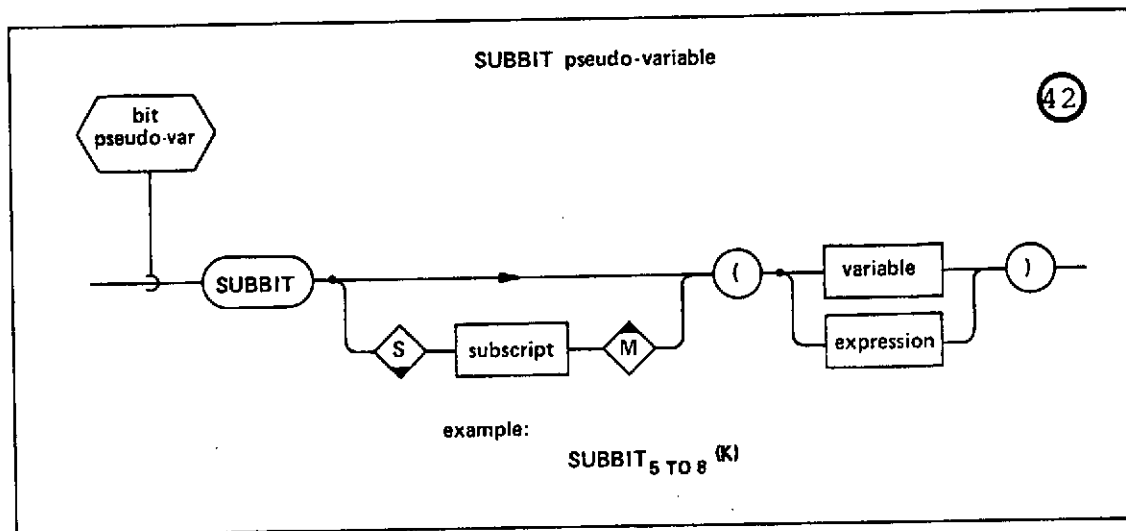
result is decimal representation of the bit pattern of the argument

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

6.5.4 The SUBBIT Pseudo-variable.

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

SYNTAX:



SEMANTIC RULES:

1. The keyword SUBBIT denotes the pseudo-variable.
2. SUBBIT has one argument only. If it appears in an assignment context, the argument must be a <variable>. If it appears as an operand of a bit expression, the argument must be an <expression>.
3. The argument may be of integer, scalar, bit or character type, and may optionally be arrayed.
4. The effect of SUBBIT is to make its argument look like an operand of bit type. (If the argument is arrayed, then it looks like an arrayed bit operand.)
5. <subscript> represents component subscripting upon the pseudo-variable. It has the same semantic meaning as if it were subscripting a bit variable (see Section 5.3.5).

6. The length of the argument in bits may in some implementations be greater than the maximum length of a bit operand. Let the maximum length of a bit operand be N bits. If SUBBIT is unsubscripted, only the N leftmost bits of the machine representation of the data-type of the argument are visible. If the representation is less than N , the number of bits visible is equal to the length of the particular data argument.
7. Partitioning subscripts of SUBBIT may make between 2 and N bits from the representation of the argument type visible at any time (i.e. the partition size is $\leq N$.) The partition size must be known at compile time. If the representation is less than the specified partition size, binary 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;
```

```
⋮
```

```
SUBBIT33 TO 64(P)
```

```
bits 33 through 64 of the machine
representation of P look like a 32-bit
bit variable
```

```
bits 1 through 32 are invisible
```

6.5.5 Summary of Argument Types.

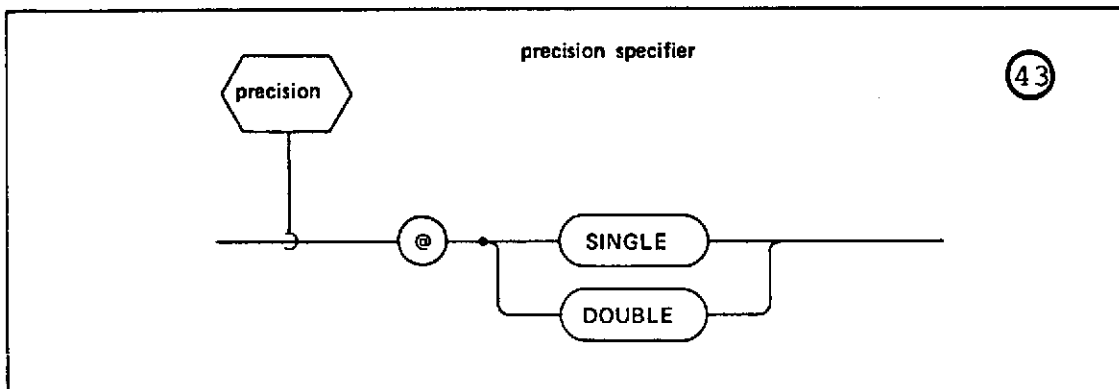
The checkmarks in the following table indicate the legal argument types for each conversion function.

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

6.6 Explicit Precision Conversion.

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

SYNTAX:



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.

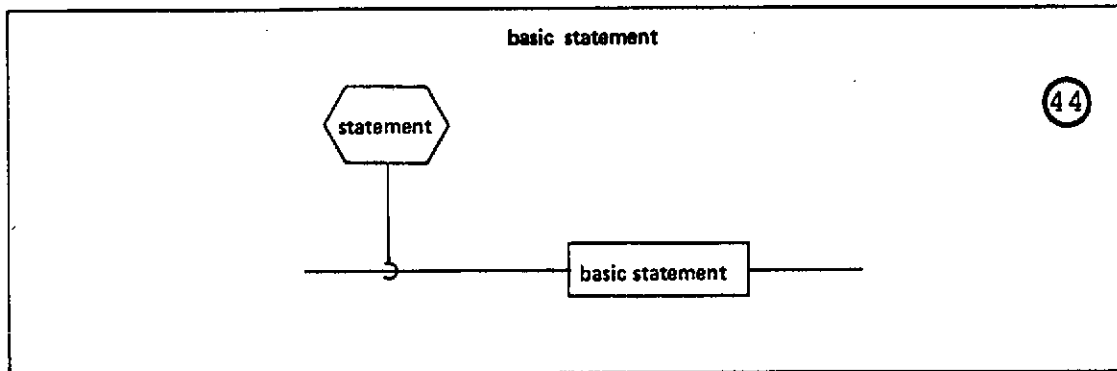
7. EXECUTABLE STATEMENTS

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

7.1 Basic Statements.

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

SYNTAX:

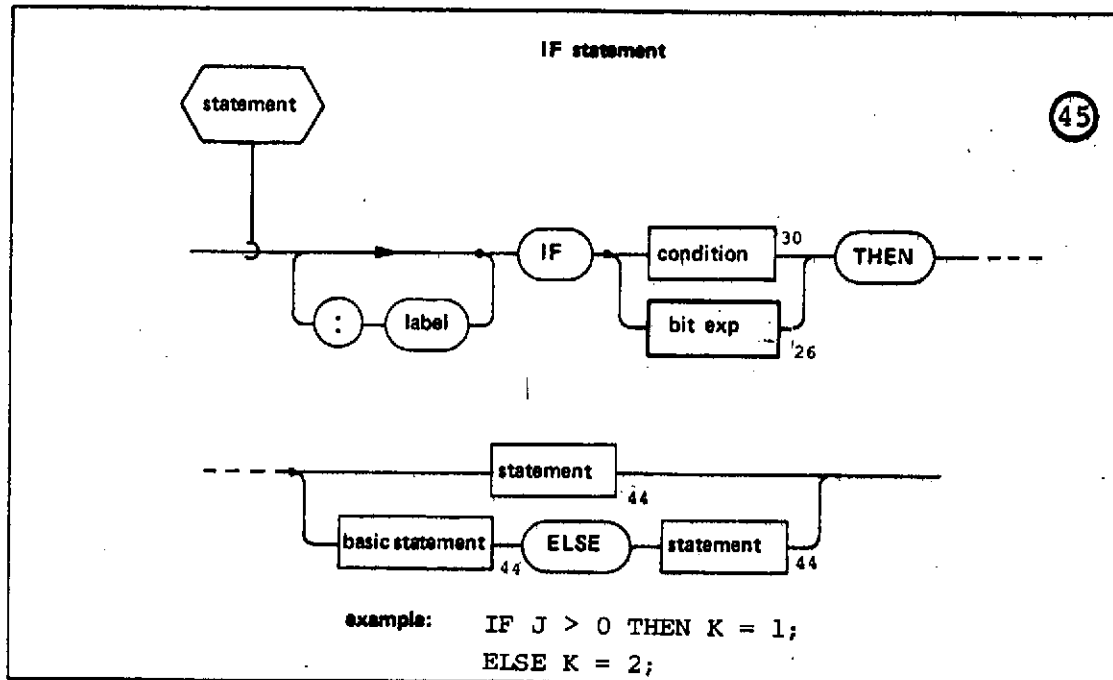


Any <basic statement>, unless it is imbedded in an IF statement or ON ERROR statement, may optionally be labelled with any number of <label>s. Not all forms of <basic statement> are described in this Section. Real time programming statements are described in Section 8, error recovery statements in Section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

7.2 The IF Statement.

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

SYNTAX:



SEMANTIC RULES:

1. The IF statement, unless it is imbedded in another IF statement or in an ON ERROR statement, may optionally be labelled with any number of <label>s.
2. The option to label the <statement> or <basic statement> of an IF statement is not disallowed. However, such labels may only be referenced by REPEAT or EXIT statements within the (compound) <statement> or <basic statement> thus labelled.
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.

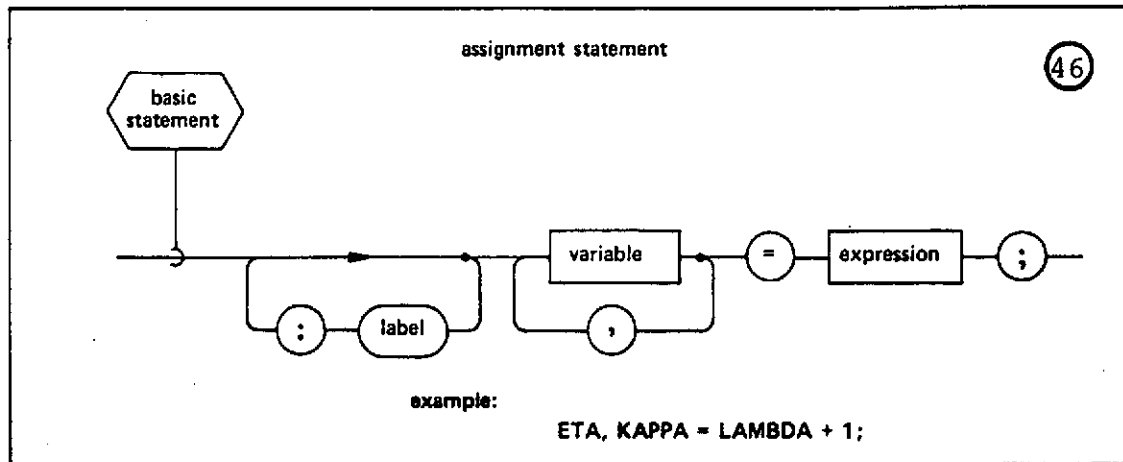
5. If the <condition> or <bit exp> is FALSE then the <statement> or <basic statement> following the keyword THEN is not executed. If the ELSE clause is present then the <statement> following the keyword ELSE is executed instead, and then execution proceeds to the <statement> following the IF statement. 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. <variable> may not be an event variable or an input parameter of a procedure or function block.
2. The effective order of execution of an assignment statement is as follows:
 - any subscript expressions on the left-hand side are evaluated;
 - the right-hand side <expression> is evaluated;
 - the values of the left-hand side <variable>s are changed.
3. If the <expression> on the right-hand side is arrayed, then all the <variable>s on the left-hand side must be arrayed. The number of dimensions of arrayness on each side must be the same, and corresponding dimensions on either side must match in size.

4. If the <expression> on the right-hand side is not arrayed then it is still possible for one or more <variable>s on 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:
 - Assignment of an integer <expression> to a scalar <variable> is allowed. Depending on the implementation this may cause loss of decimal places of accuracy.
 - 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 zeroes if it is too short.
2. The effect of a left-hand side <variable> being a <bit pseudo-var> is described in Section 6.5.4.

SEMANTIC RULES (character assignments):

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

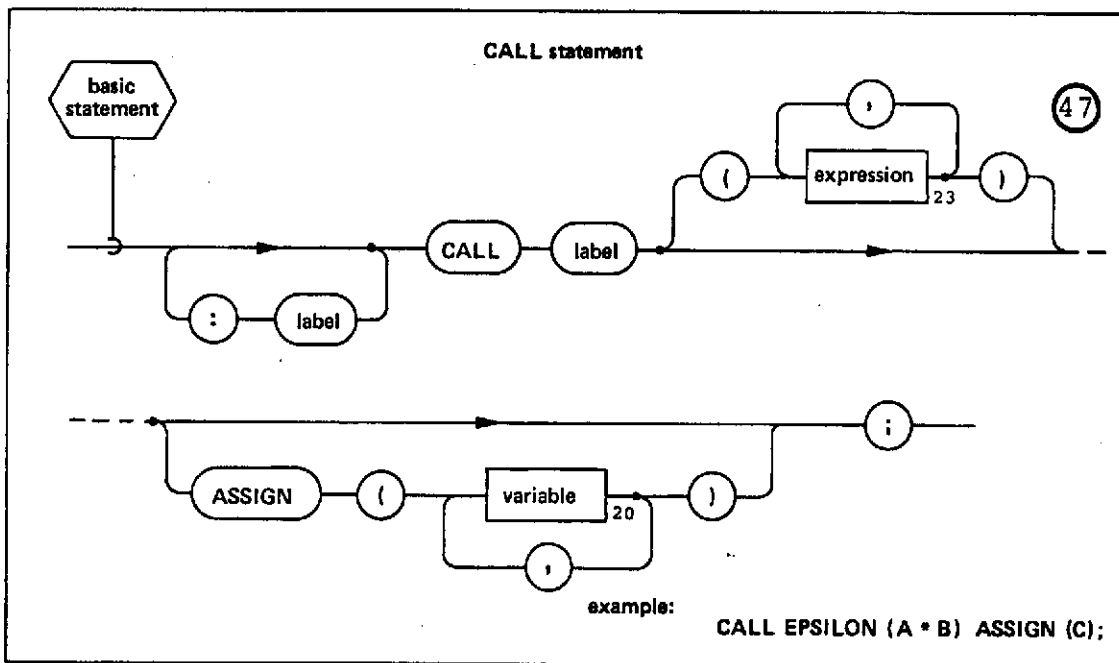
SEMANTIC RULES (structure assignments):

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

7.4 The CALL Statement.

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

SYNTAX:



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:
 - No component subscripting for bit and character types.
 - If component subscripting is present, <variable> must be subscripted so as to yield a single (unarrayed) element of the <variable>.
 - If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.
6. Assign arguments are "call-by-reference". Input arguments are either "call-by-reference" or "call-by-value"¹.
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:
 - If it is of lock group N, then the corresponding assign parameter must be of lock group N, or *.
 - If it is of lock group *, then the corresponding parameter must also be of group *.

¹ 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.

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

- precisions need not match;
- the lengths of bit arguments need not match;
- the lengths of character arguments need not match;
- implicit integer to scalar and scalar to integer conversions are allowed;
- implicit integer and scalar to character conversions are allowed;
- 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.

10. 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.

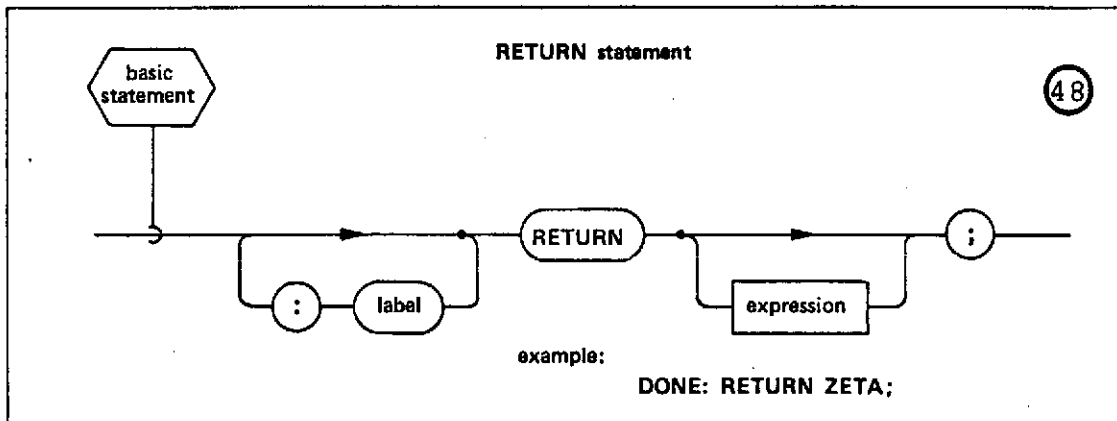
example:

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


7.5 The RETURN Statement.

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

SYNTAX:



GENERAL SEMANTIC RULES:

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

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

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

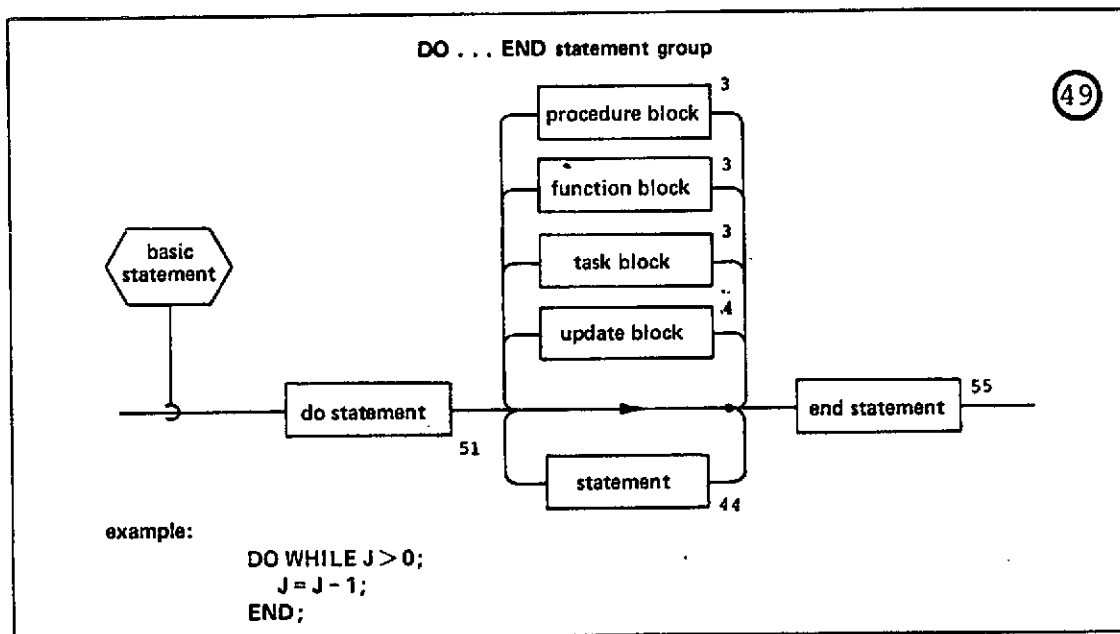
4. <expression> must always appear in RETURN statements of <function block>s. Execution must always end on logically reaching a RETURN statement of such a block, and not by logically reaching the delimiting CLOSE statement.

124

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:



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, TASK, or UPDATE blocks. The form of the <do statement> determines how the <statement>s within the group are executed.

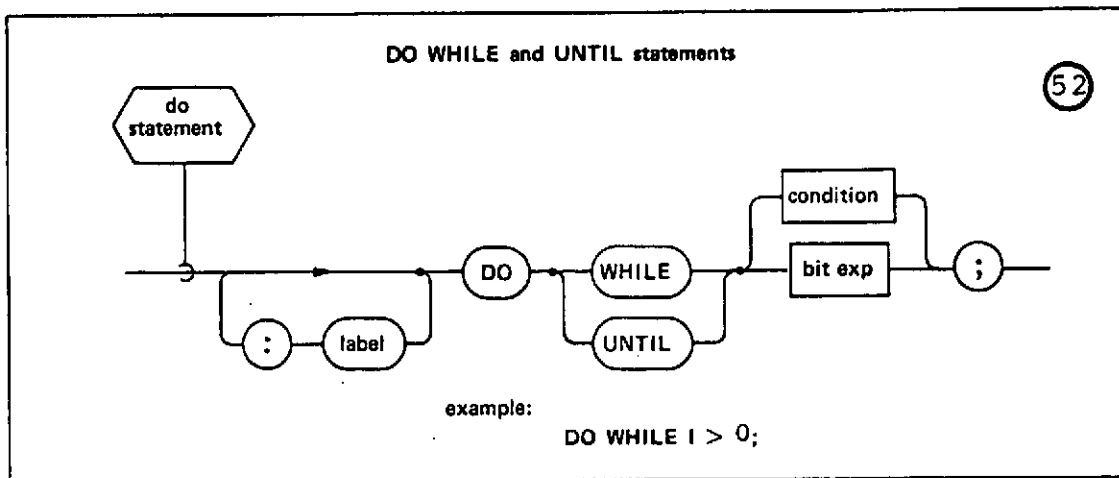
Page intentionally left blank

Page intentionally left blank

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:

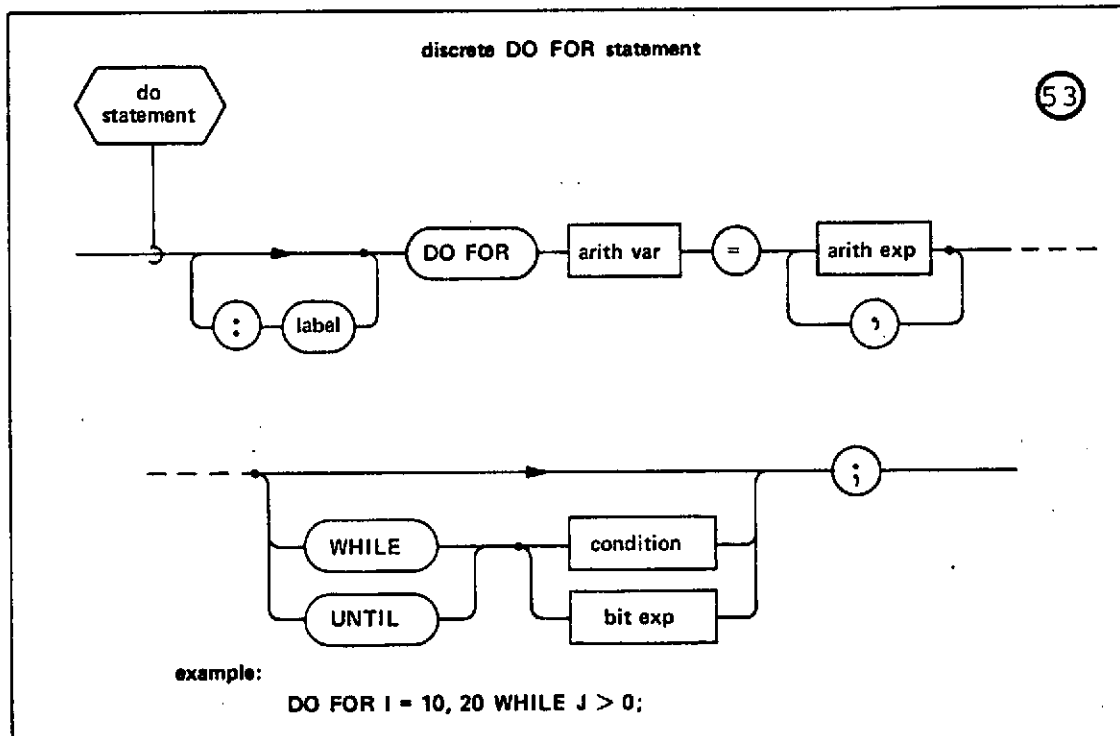
1. There is no semantic restriction on <condition>. <bit exp> must be boolean and unarrayed (i.e., of 1-bit length). The <condition> or <bit exp> is re-evaluated every time the group of <statement>s is executed.
2. In the DO WHILE version, the group of <statement>s is repeatedly executed until the value 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:

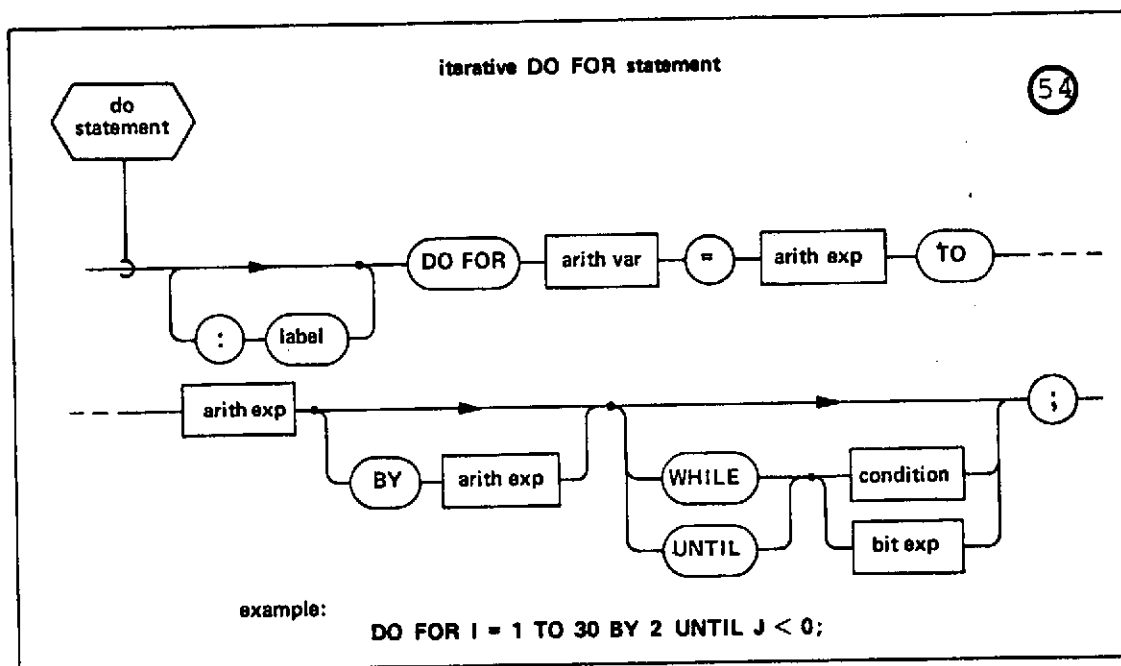
1. <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable.
2. The maximum number of times of execution of the group of <statement>s is the number of <arith exp>s in the assignment list.
3. <arith exp> is an unarrayed integer or scalar expression.

4. At the beginning of each cycle of execution of the group the next <arith exp> in the list (starting from the left-most) 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>. <bit exp> must be boolean and unarrayed (i.e. of 1-bit length). The <condition> or <bit exp> is re-evaluated 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:

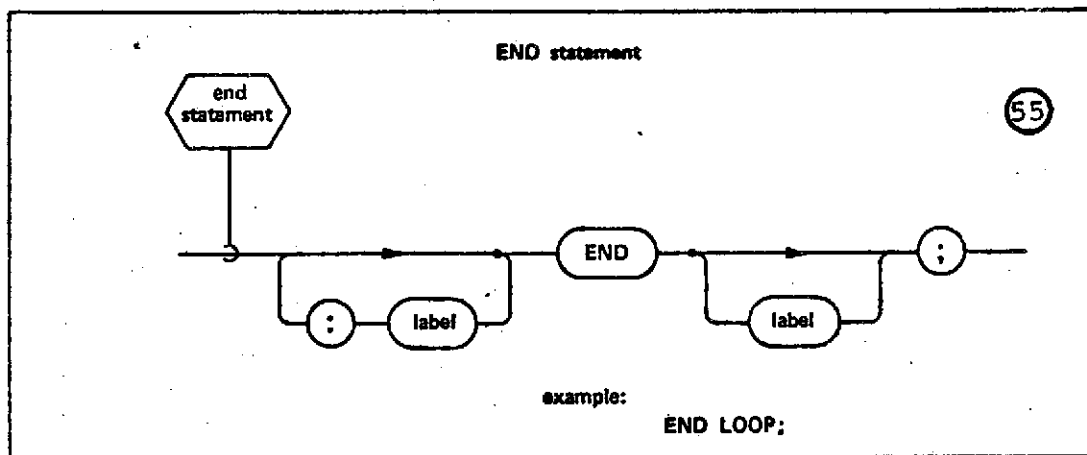
1. <arith var> is the loop variable of the DO FOR statement. It may be any unarrayed integer or scalar variable.
2. Each <arith exp> is any unarrayed integer or scalar expression. All are evaluated prior to the first cycle of execution of the group.

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 k^{th} cycle of execution is equal to its value on the $K-1^{\text{th}}$ 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:



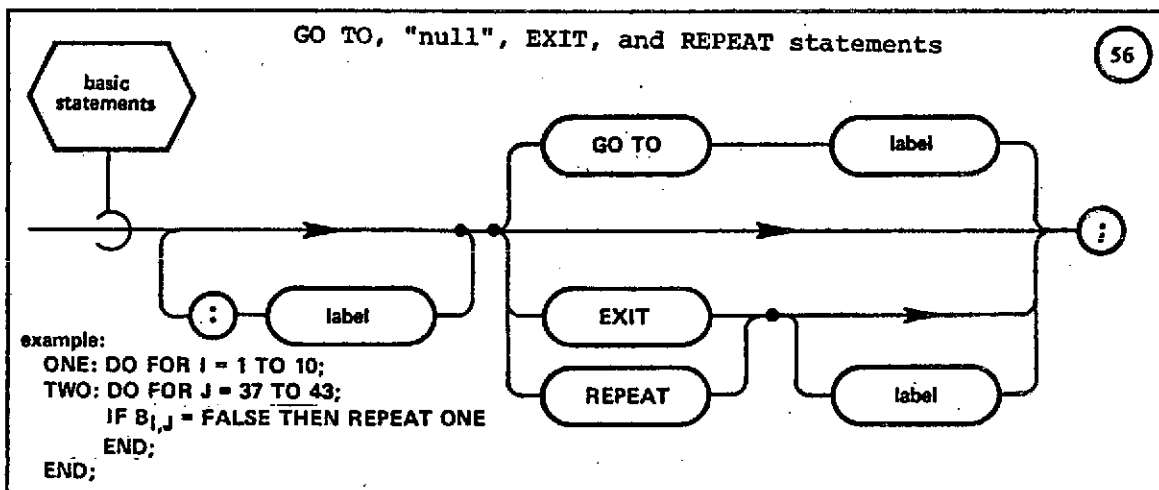
SEMANTIC RULES:

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:



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.

8. REAL TIME CONTROL

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

8.1 Real Time Processes and the RTE.

In HAL/S, a program or task may be scheduled as a process and placed in the process queue. Although the process created is given the same name as the program or task, it is important to distinguish the static PROGRAM or TASK block from the dynamic program or task process created. Two processes are actually involved in the creation of a process: the scheduling process, or "father"; and the scheduled process, or "son".¹

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

Each process in the RTE's process queue is at any instant in one of a number of states. For the purposes of this Section, the following states are defined:²

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

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

¹ except of course for the first or "primal" process which must be created by the RTE itself.

² these states are not necessarily definitive of those actually existing in any particular implementation of the RTE.

Execution of a CLOSE or RETURN statement by an active process causes termination of the process and return to the RTE. In the case of a process which has schedule dependent processes, such execution places the father in a stall state until the sons have all terminated before the process is itself terminated.

124

8.2 Timing Considerations.

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

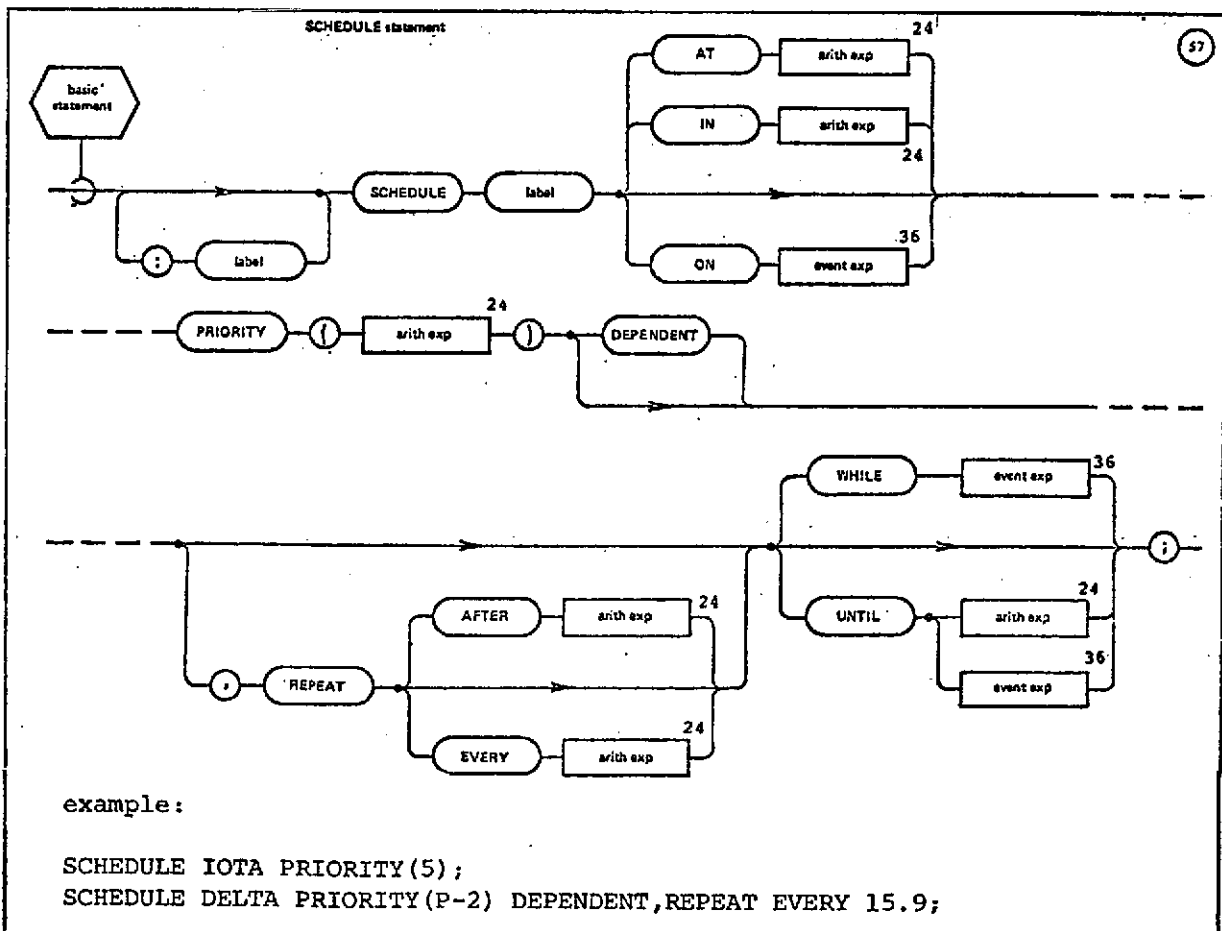
8.3 The SCHEDULE Statement.

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

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

76 |

SYNTAX:



76 |

SEMANTIC RULES:

1. SCHEDULE <label> schedules a program or task with the name <label>, placing a new process with name <label> in the process queue. A run time error results if a process of that name already exists in the process queue. Unless otherwise specified the RTE puts the new process in the ready state immediately after execution of the SCHEDULE statement.
2. The phrase IN <arith exp> is used to cause the process to be put in the stall state for a fixed RTE-clock duration. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then the process is put immediately in the ready state.
3. The phrase AT <arith exp> is used to cause the process to be put in the stall state until a fixed RTE-clock time. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than the current RTE-clock time, and the REPEAT EVERY option is not specified, then the process is put immediately in the ready state. If the value is less than the current RTE time and the REPEAT EVERY option was specified, then phased scheduling takes place. The process is put in a stall state until a future time computed by the expression $CT + RE - ((CT - AT) \text{MOD } RE)$, where CT = current time, RE = REPEAT EVERY cycle time, and AT = originally specified AT time. 125
4. The phrase ON <event exp> is used to cause the process to be put in the stall state until some event condition is satisfied. Starting from the time of execution of the SCHEDULE statement, the <event exp> is evaluated at each "event change point"³ until its value becomes TRUE. At that time the process is placed in the ready state. If the value of <event exp> is TRUE upon execution of the SCHEDULE statement, then the process is immediately put in the ready state. 141

³ the meaning of an "event change point" is defined in Section 8.8.

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

10. The UNTIL <arith exp> phrase specifies a cancellation criterion based on RTE-clock time. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. For any process, cyclic or non-cyclic, the following is true. If the value of <arith exp> is not greater than the current RTE-clock time, then the process is never entered in the process queue. Otherwise if the value of <arith exp> becomes equal to the RTE-clock time before the process is initiated, then the process is removed from the process queue. If neither of the above is true, then the following ensues. If the process is non-cyclic, the phrase has no further effect. If the process is cyclic, then cancellation may occur in one of two ways. If the value of <arith exp> becomes equal to the RTE-clock time while the process is in an inter-cycle stall state, the process is cancelled immediately. If it happens during a cycle, the process is cancelled immediately on completion of the current cycle. The next cycle proceeds if cancellation does not occur.

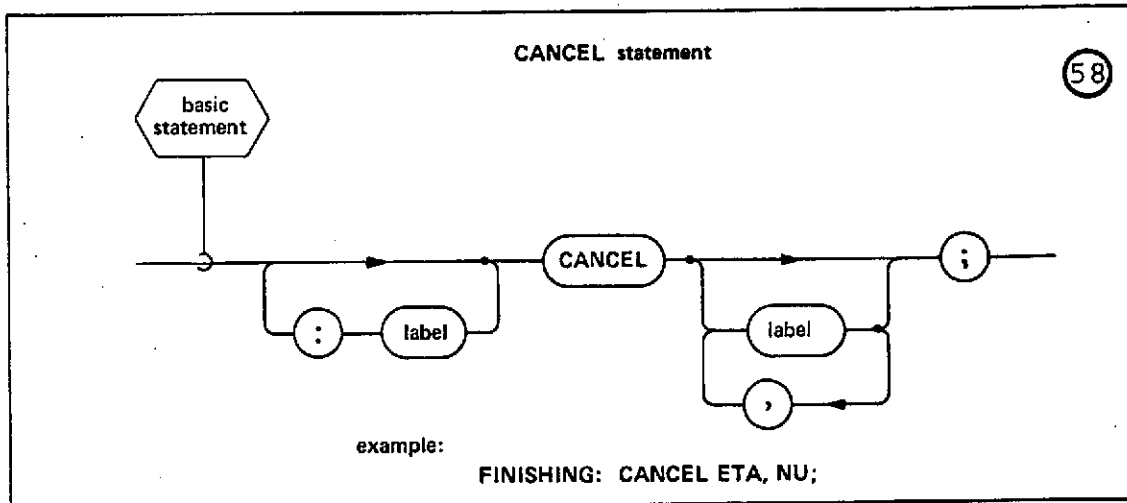
11. The WHILE <event exp> phrase specifies a cancellation criterion based on an event condition. For any process, cyclic or non-cyclic, the following is true. If the value of <event exp> is FALSE at the time of execution of the SCHEDULE statement, then the process is never placed in the process queue. If not, then <event exp> is evaluated at every "event change point" until its value becomes FALSE. If it becomes FALSE before the process is initiated, then the process is removed from the process queue. If neither of the above is true, then the following ensues. If the process is non-cyclic, the phrase has no further effect. If the process is cyclic, then cancellation may occur in one of two ways. If <event exp> becomes FALSE at any "event change point" occurring while the process is in an inter-cycle stall state, the process is cancelled immediately. If it occurs during the cycle, the process is cancelled immediately on completion of the current cycle. If cancellation does not occur, the next cycle proceeds.

12. The UNTIL <event exp> phrase also specifies a cancellation criterion based on an event condition. However, it differs fundamentally from the WHILE <event exp> phrase in that it always allows at least one cycle of a cyclic process to be executed. Consistent with this, the phrase has no meaning and therefore no effect in the case of a non-cyclic process. For a cyclic process, the value of the <event exp> is evaluated at every "event change point" from the time of execution of the SCHEDULE statement. Beginning with the end of the first cycle of execution, cancellation may occur in one of two ways. If the <event exp> becomes TRUE at any "event change point" occurring while the process is in an inter-cycle stall state, the process is cancelled immediately. If it occurs during a cycle, the process is cancelled immediately on completion of the current cycle. If cancellation does not occur, the next cycle proceeds.

8.4 The CANCEL Statement.

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

SYNTAX:



SEMANTIC RULES:

1. CANCEL <label> causes cancellation of the process <label> and its dependent processes. A run time error results if the process queue contains no process with that name.⁴ The CANCEL statement can be used to cancel any number of processes simultaneously. 106
2. If the CANCEL statement has no <label>, cancellation of the process executing the CANCEL statement and its dependents is implied. 106

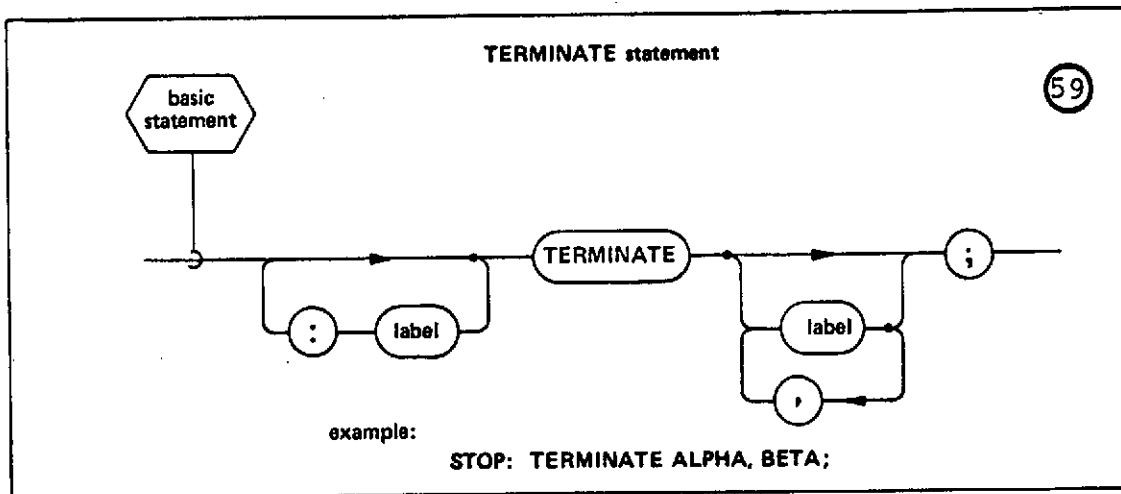
⁴ the default action taken by the Error Recovery Executive for this and other similar errors may be to ignore the error.

3. If at the time of execution of the CANCEL statement, a process to be cancelled has not yet been initiated, then the process is merely removed from the process queue. This applies to both cyclic and non-cyclic processes.
4. If at the time of execution of the CANCEL statement a process to be cancelled has already been initiated, then the following ensues. If the process is non-cyclic and it has already been initiated, the CANCEL statement has no effect. If the process is cyclic, then the process is cancelled at the end of the current cycle of execution.
5. A cancelled process with dependent offspring is put in the stall state until its dependent offspring are terminated. At that time, the parent process is terminated.

8.5 The TERMINATE Statement.

The termination of a process implies the immediate⁵ cessation of execution of the process and all its dependent sons, and their removal from the process queue. The TERMINATE statement is used to direct the RTE to terminate specified processes.

SYNTAX:



SEMANTIC RULES:

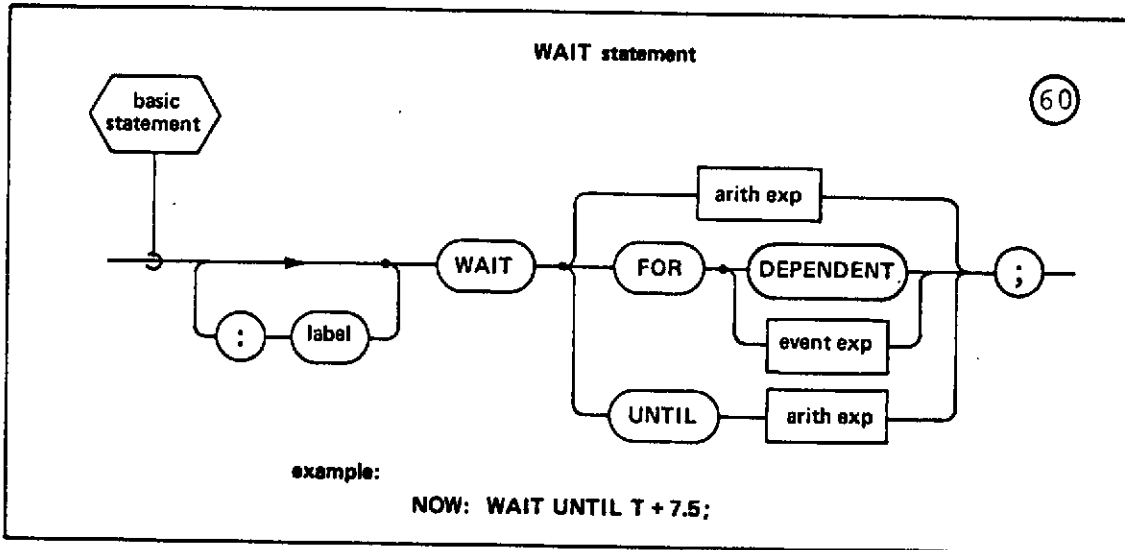
1. TERMINATE <label> causes termination of the process <label>. A run time error results if a process of that name is not in the process queue, or if it is not a dependent son of the process currently executing the TERMINATE statement. The TERMINATE statement can be used to terminate any number of processes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

⁵ subject of course to implementation dependent safety constraints.

8.6 The WAIT Statement.

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

SYNTAX:



SEMANTIC RULES:

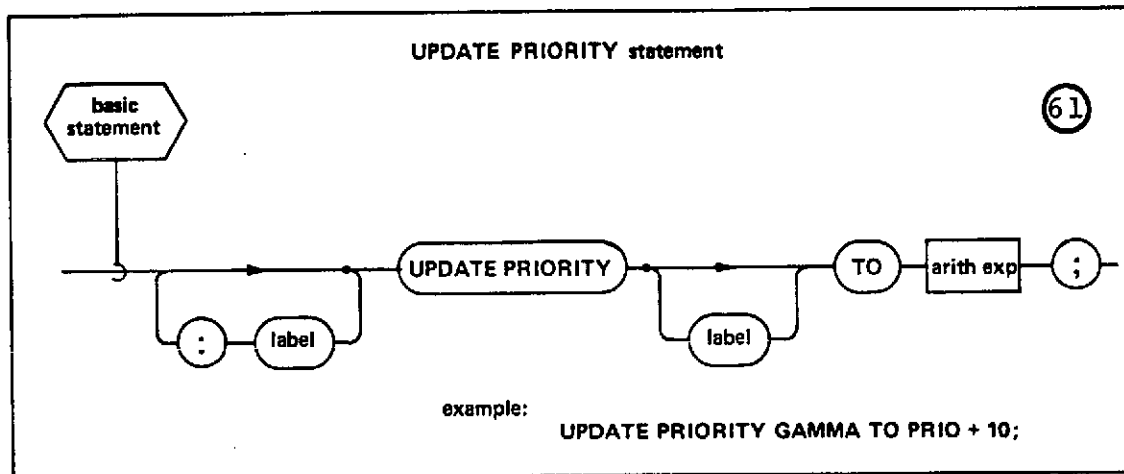
1. The WAIT <arith exp> version specifies that the process executing the WAIT statement is to be placed in the stall state for an RTE-clock duration fixed by the value of the expression. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the WAIT statement.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

3. The WAIT FOR DEPENDENT version specifies that the process executing the WAIT statement is to be placed in the stall state until all its dependent sons have terminated. If there are no such processes, the WAIT statement has no effect.
4. The WAIT FOR <event exp> version specifies that the process executing the WAIT statement is to be placed in the stall state until an event condition is satisfied. Starting from the time of execution of the WAIT statement, the <event exp> is evaluated at every "event change point" until its value becomes TRUE, whereupon the process is returned to the READY state. If the value of <event exp> is TRUE upon execution of the WAIT statement, then the statement has no effect.

8.7 The UPDATE PRIORITY Statement.

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

SYNTAX:



SEMANTIC RULES:

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

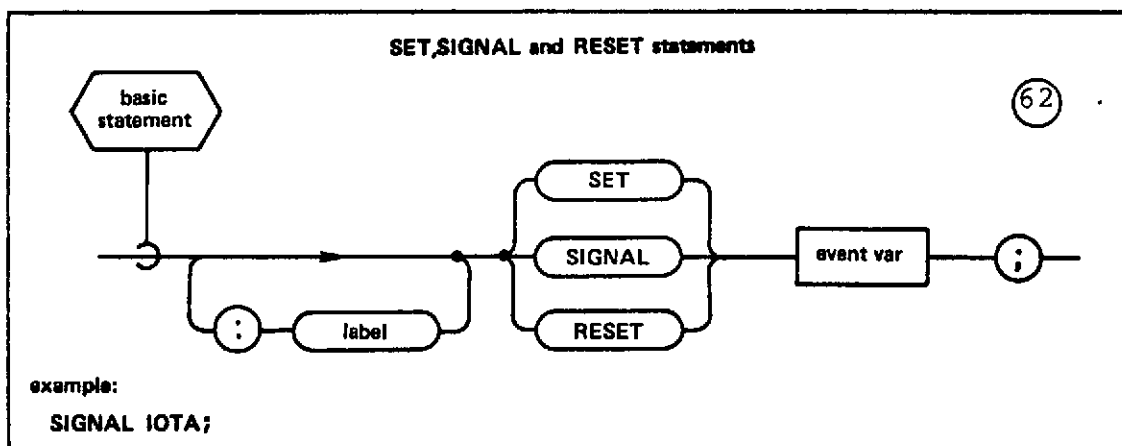
8.8 Event Control

Although a formal specification of events and event expressions has already been 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. Conceptually the two forms of HAL/S events (latched and unlatched) may be thought of as the software counterparts of hardware discretes and timing lines, respectively.

- a latched event may be thought of as a boolean system state which may be SET or RESET by appropriate actions, or momentarily changed for signalling purposes.
- an unlatched event may be thought of as the software counterpart of a timing line which is used purely for signalling - it is normally FALSE but becomes TRUE momentarily when a signal action is executed.

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

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



GENERAL SEMANTIC RULE:

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

SEMANTIC RULES (latched <event var>s):

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

SEMANTIC RULES (unlatched <event var>s):

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

SUMMARY:

		SET	RESET	SIGNAL
unlatched event		illegal	illegal	Take all event actions depending on TRUE state of <event var>
latched event	old value is FALSE	<ol style="list-style-type: none"> 1. Set event state to TRUE 2. Take all event actions depending on TRUE state of <event var> 	no action	Take all event actions depending on TRUE state of <event var>
latched event	old value is TRUE	no action	<ol style="list-style-type: none"> 1. Set event state to FALSE 2. Take all event actions depending on FALSE state of <event var> 	Take all event actions depending on FALSE state of <event var>

8.9 Process-events.

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

8.10 Data Sharing and the UPDATE Block.

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

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

OPERATIONAL RULES:

1. If two processes both require variables from the same lock group to be changed, then the first process entering its **UPDATE** block must complete execution of the block before the other process can enter its own **UPDATE** block. The second process is placed in a stall state for the duration.
2. If one process entering an **UPDATE** block requires a variable(s) with the attribute **LOCK(*)** to be changed, then the situation is equivalent to one in which the process requires use of a variable from every lock group. 124
3. If only one of the processes requires a variable of a lock group to be changed, the other merely requiring it to be accessed, then depending on the implementation, either Rule 1 or 2 holds, or some overlap in execution of the two processes' **UPDATE** blocks is allowed. The nature of such overlap must be such as to provide exclusive use of the lock group by the process requiring its change between the point where the variable is changed and the close of the **UPDATE** block.
4. If both processes only require a variable of the same lock group accessed, then execution of the two processes' **UPDATE** block may be allowed to overlap depending upon implementation.
5. If there are several simultaneous conflicts in using shared variables because of the participation of more than two processes, or more than one lock group, then the most restrictive of Rules 1 through 4 required is applied to resolve the conflicts.

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/S implementation possesses a unique collection of such errors. The errors in the collection are said to be "system-defined". In any implementation every possible system-defined error is assigned a unique "error code". In addition, a number of other legal error codes not assigned to system-defined errors may exist. These can be used by the HAL 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 consist 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.

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

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

E

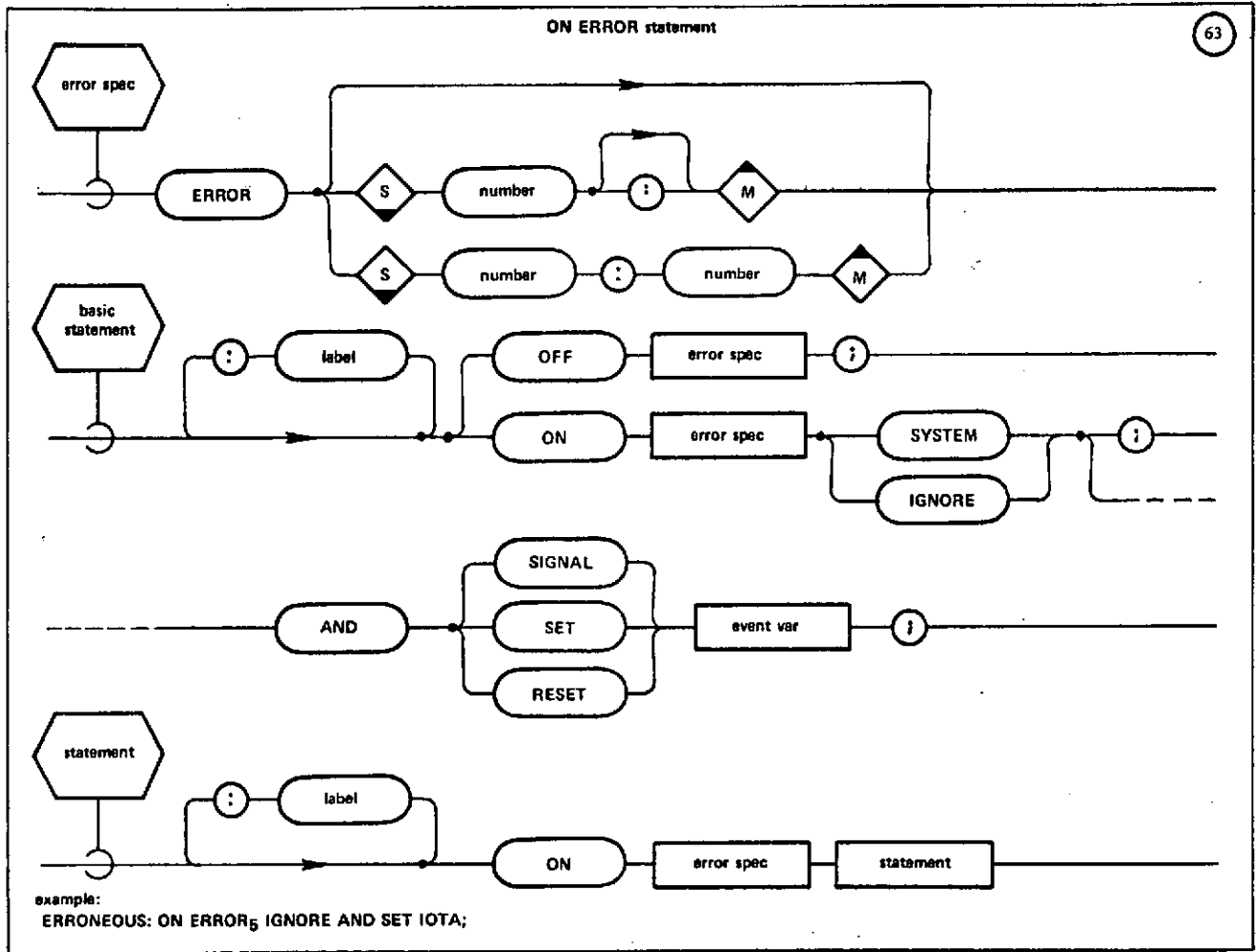
9.1 The ON ERROR Statement.

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/S 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 happen:

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

SYNTAX:



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.
 - The form of <error spec> without subscript is used to specify all error groups.
 - The subscript construct <number> with optional following colon is used to specify a selected <error group>. The value of <number> is restricted to the set of error group numbers defined for a particular implementation.
 - The subscript construct <number>: <number> is used to specify a selected error code. The leftmost <number> designates the error group number; the rightmost <number> the selected error number within the group. Values are restricted to the set of error codes defined for a particular implementation.
3. The form ON ERROR specifies the modification of the error recovery actions for the given <error spec>. OFF ERROR specifies the removal of a modification previously activated in the 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 optionally be labelled. However, such labels may only be referenced by EXIT or REPEAT statements within the (compound) <statement> thus labelled. 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> whilst other forms of ON ERROR are <basic statement>s. The form ON ERROR ... <statement> may therefore not be the true part of an IF...THEN...ELSE statement.
8. If an ON ERROR possesses a SYSTEM or IGNORE clause, it may also possess an additional SIGNAL, SET, or RESET clause. The purpose is to cause the value of an <event var> to be changed on the occurrence of a specified error. Its semantic rules are the same as those described for the corresponding SIGNAL, SET and RESET statements in Section 8.8. Note that if <event var> contains a subscript expression, then that expression will be evaluated at the time of execution of the ON ERROR statement, not on the occurrence of the error.

PRECEDENCE RULE:

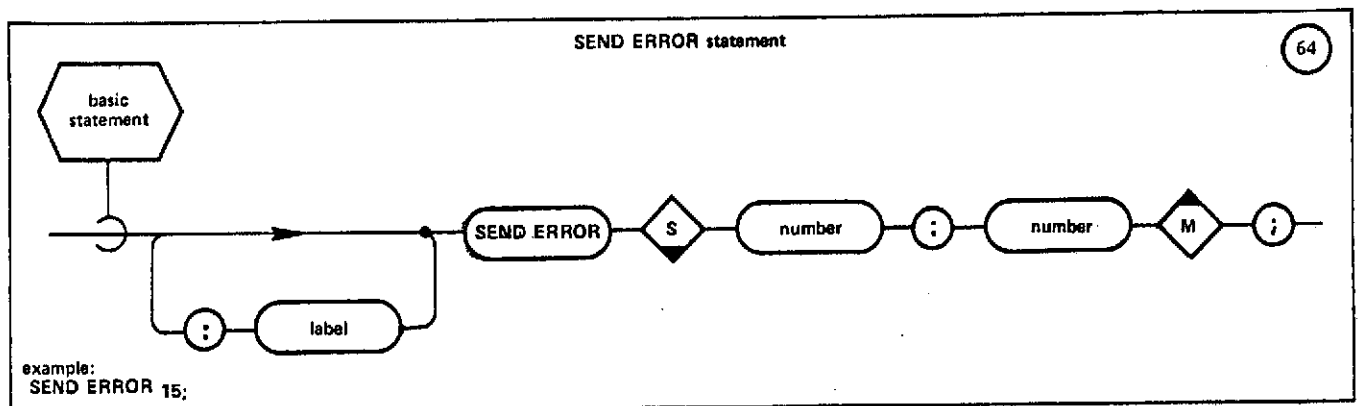
In a code block the action specified by an ON ERROR is only superseded by another if the two <error spec>s are of identical form. Similarly an OFF ERROR nullifies the effect of a previous ON ERROR only if the two <error spec>s are of identical form. However, different forms of <error spec > may involve the same error group or error code. It is logically possible for up to three ON ERRORS, 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
all groups	-	LAST 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. <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/S language provides for two forms of I/O: sequential I/O with conversion to and from an external character string representation; and random-access record-oriented I/O.

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

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

10.1 Sequential I/O Statements.

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

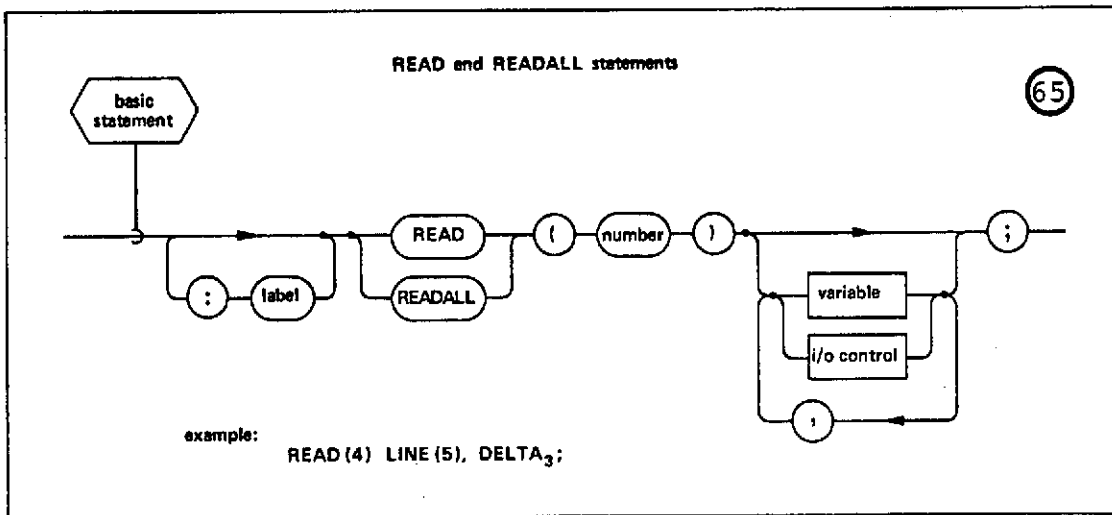
HAL/S pictures the physical device as moving across the file a read or write "device mechanism" which actually performs the data transfer. The device mechanism has at every instant a definite column and line position on the file. The action of transmitting one character to or from the file is followed 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/S sequential I/O statements are the READ, READALL, and WRITE statements. Within these statements I/O control functions can be used to cause explicit positioning of the device mechanism on the file.

10.1.1 The READ and READALL Statements.

The sequential input of data is accomplished in HAL/S by employing either a READ or 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. <number> is any legal I/O channel number.
2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.
3. Unless overridden by explicit <i/o control> 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. <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. <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/S 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:
 - The current variable element is left unchanged;
 - All remaining <variable>s in the statement are unchanged;
 - All remaining control functions in the statement are ignored.

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

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

- preceded by a comma or semicolon;
- preceded by one or more blanks following the last comma or semicolon.

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

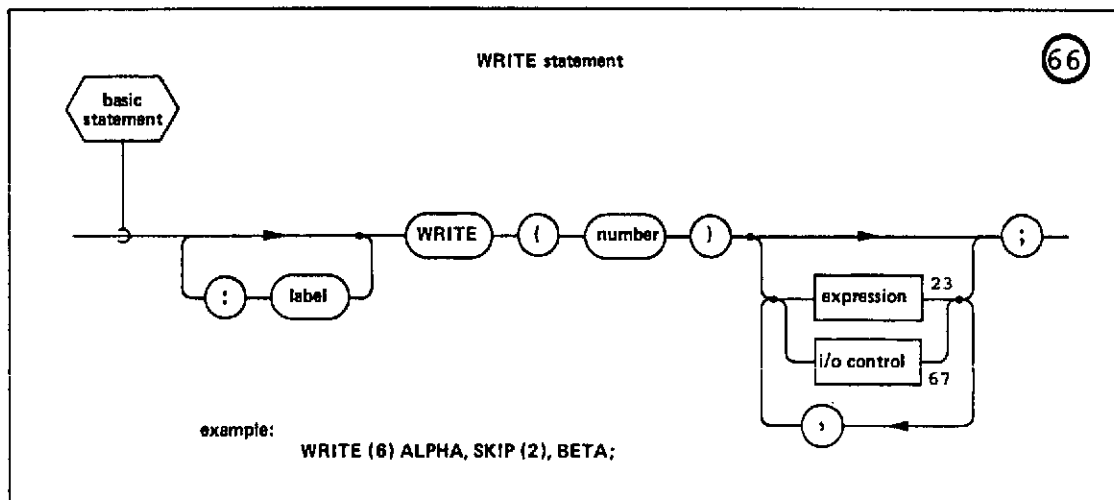
6. Fields are assumed to be in a standard external format matching the type of each corresponding type of variable element. A list of standard external formats is given in Appendix E. A mismatch between standard external format and element type causes a run time error.

111

10.1.2 The WRITE Statement.

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

SYNTAX:



SEMANTIC RULES:

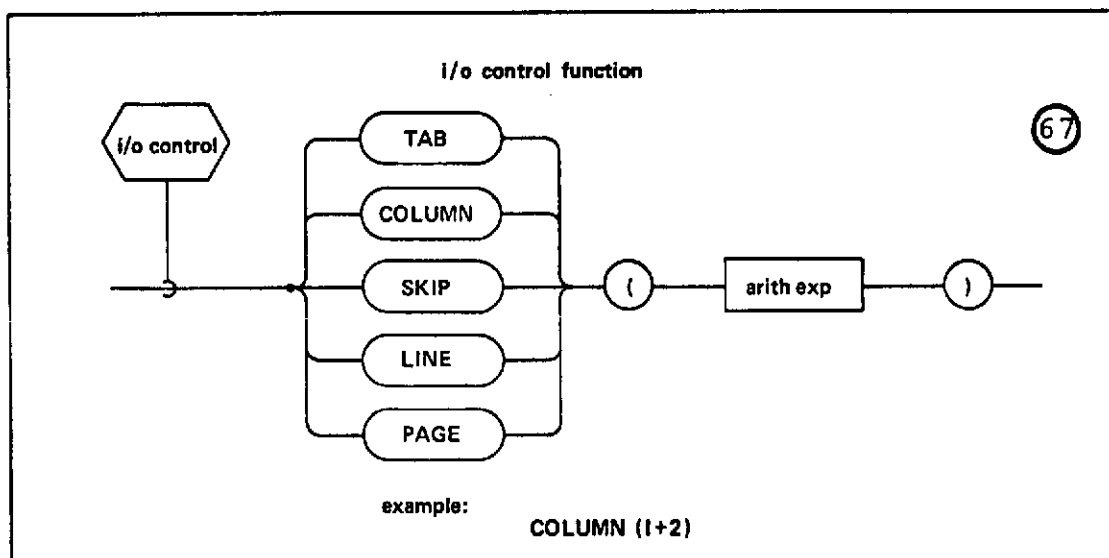
1. <number> is any legal I/O channel number.
2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.
3. There are no semantic restrictions on <expression>.
4. If <expression> is of vector or matrix type, or is an array or structure, then each element thereof is transmitted sequentially in its "natural sequence".

5. Unless overridden by explicit <i/o control> 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. A list of standard external formats is given in Appendix E.
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:

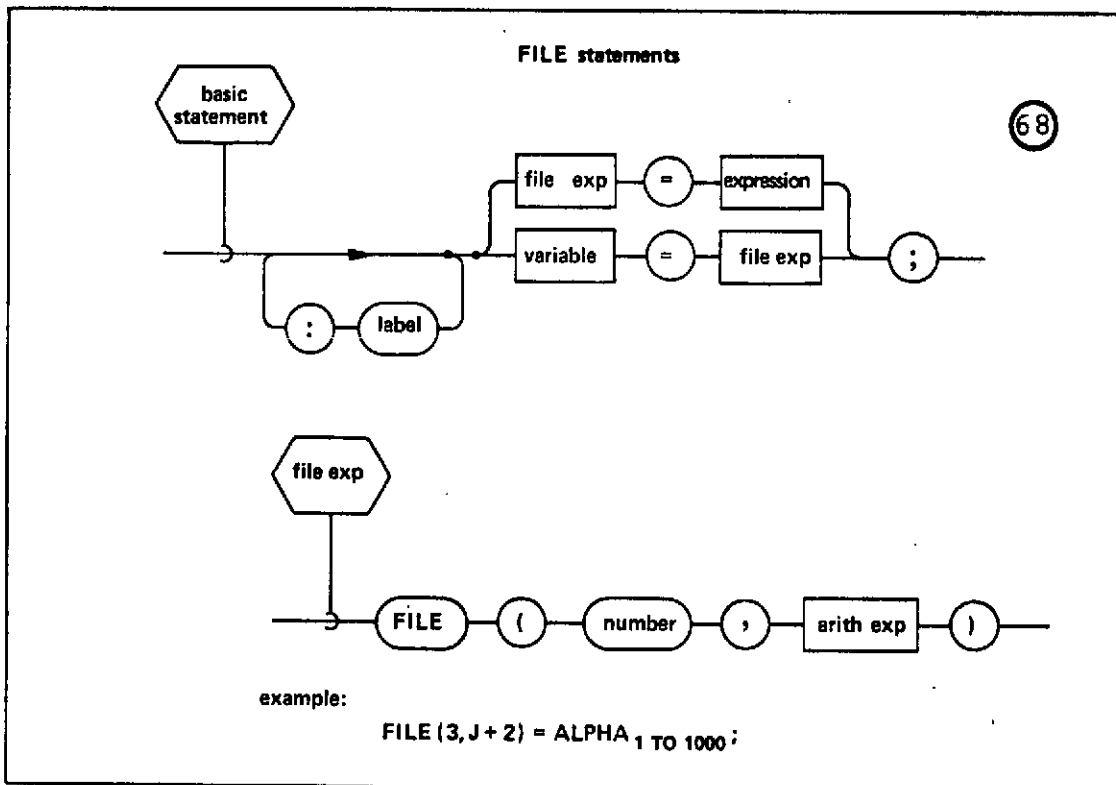
1. <arith exp> is an unarrayed scalar or integer arithmetic expression specifying a value to the control function. The value is treated as an integer: scalar values are rounded to the nearest integer prior to use. In the following rules, let the value of <arith exp> be denoted by K .
2. TAB (K) specifies 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 column K of the current line. Values of K may range from 1 to an implementation dependent maximum value. Column indices outside the legitimate range cause 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 unpagged.
 - 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.
 - Unpagged 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 backwards movement may be illegal.
6. PAGE (K) is only applicable to paged files and specifies page movement relative to the current page. If K is positive the movement is forward, towards 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.

10.2 Random Access I/O and the FILE Statement.

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

SYNTAX:



SEMANTIC RULES:

1. The statement is an output FILE statement if <file exp> is on the left of the assignment. If <file exp> is on the right, then the statement is an input FILE statement.

2. <file exp> specifies the random access I/O channel and record address to be referenced. <number> is any legal random access channel number. <arith exp> is any unarrayed integer or scalar expression. If the expression is scalar, its value is rounded to the nearest integer before use. A 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:
 - No component subscripting for bit and character types.
 - If component subscripting is present, <variable> must be subscripted so as to yield a single (unarrayed) element of the <variable>.
 - If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.
 - 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 the number of copies must be reduced to one by subscripting.
5. In the output FILE statement, there are no semantic restrictions on <expression>.
6. Compatibility between data written by an output FILE statement, and later reference to it by an input FILE statement is assumed. The exact interpretation of compatibility is implementation dependent. In general, the FILE statement transmits binary images of the internal data forms, so that compatibility will be guaranteed if the <expression> of the output FILE statement and the <variable> of the input FILE statement have the same data type and organization.

11. SYSTEMS LANGUAGE FEATURES[†]

11.1 INTRODUCTION

The systems language features of HAL/S are described in this section. The features presented here are in three sections. The new Program Organization features are "Inline Function Blocks" and "%-macros". A data-related feature of this systems language extension is the concept of "TEMPORARY variables". The NAME Facility concerns a new concept in HAL/S, the addition of NAME variables 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. Such modified diagrams are indicated by appending the small letter "s" to the diagram number.

11.2 PROGRAM ORGANIZATION FEATURES

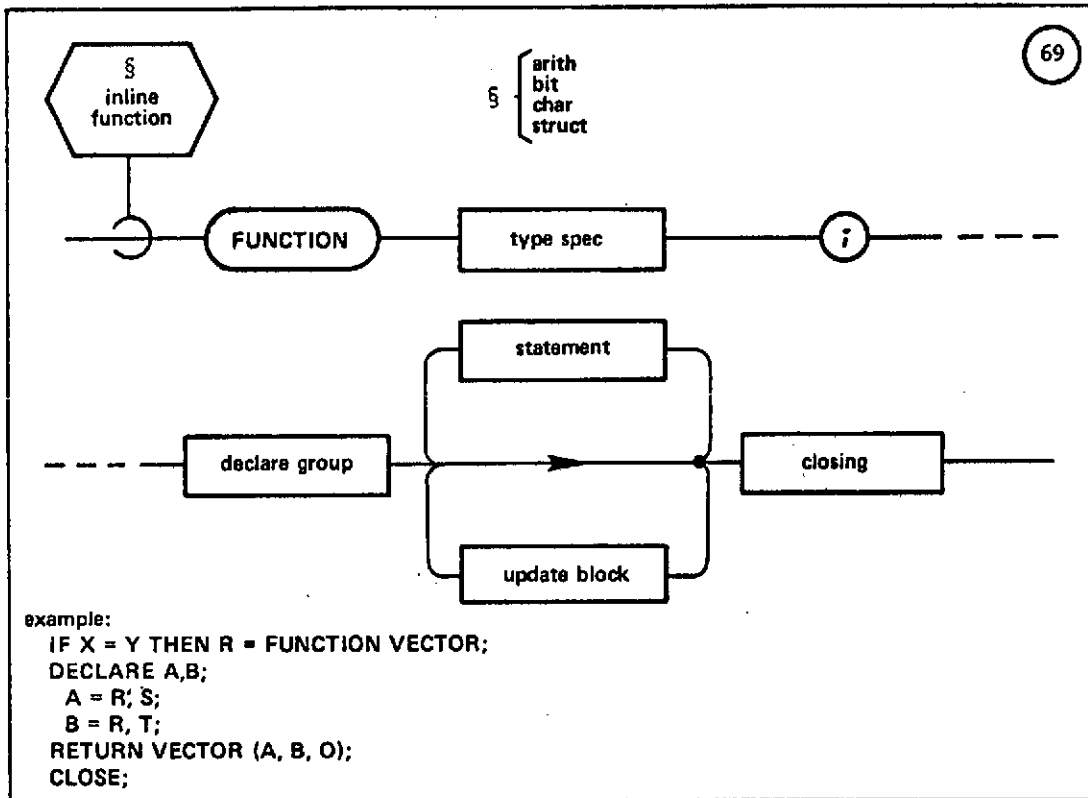
The addition of Inline Function Blocks and "%-macros" to HAL/S extends the information presented in Section 3 concerning program organization. Inline functions are a modified kind of user function in which invocation is simultaneous with block definition. %-macros may be viewed as a class of special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

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

11.2.1 Inline Function Blocks

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

SYNTAX:



SEMANTIC RULES:

1. The syntactic form is actually equivalent to that of a function block except that:
 - a) The <§inline function> has no label;
 - b) The <§inline function> has no parameters;
 - c) The <§inline function> definition becomes an operand in an expression.

2. The semantic rules for an `<$inline function>` block definition are the same as those for the `<function block>` definition described in Section 3.3, subject to restrictions listed below.
3. A `<$inline function>` may not contain the following syntactical forms:
 - All forms of I/O statements;
 - All forms of reference to user-defined PROCEDURE and FUNCTION blocks;
 - Real Time statements.
4. A `<$inline function>` may only contain one form of nested block, the `<update block>`. The following block forms are thus excluded:
 - `<function block>` definitions;
 - `<procedure block>` definitions;
 - Further nested `<$inline function>`s.
5. In use, the following semantic restriction holds: `<$inline function>`s may not appear as operands of the subscript or exponent expressions.
6. The `<$inline function>` falls into one of the following four categories:

<code><arith inline></code>	- <code><type spec></code> specifies an inline function of an arithmetic data type: SCALAR, INTEGER, VECTOR or MATRIX.
<code><bit inline></code>	- <code><type spec></code> specifies an inline function of a bit type: BOOLEAN or BIT.

- <char inline> - <type spec> specifies an inline function of the CHARACTER data type.
- <struct inline> - <type spec> specifies an inline function with a structure type specification.

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

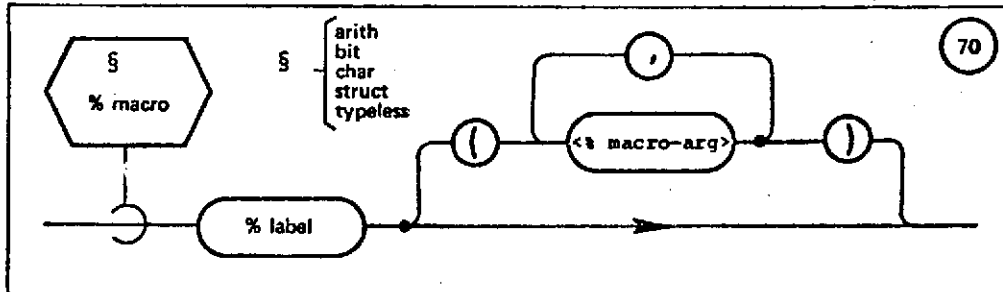
11.2.2 %-macro References

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

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

E

SYNTAX:



SEMANTIC RULES:

1. The `%-macro` reference falls into one of the following five categories based upon data type:
 - `<arith %-macro>` is a reference to a `%-macro` which returns an arithmetic value of INTEGER, SCALAR, VECTOR or MATRIX data type.
 - `<bit %-macro>` is a reference to a `%-macro` which returns a bit string value.
 - `<char %-macro>` is a reference to a `%-macro` which returns a value of the CHARACTER data type.
 - `<struct %-macro>` is a reference to a `%-macro` which returns a structure data value.
 - `<typeless %-macro>` is a reference to a `%-macro` which performs some systems function but returns no value and may only be referenced from a `<%-macro call statement>`. (See Section 11.2.4 below).

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

2. The `<%label>` is a reserved word beginning with the character `"%"` which identifies the `%-macro` in question. The character `"%"` distinguishes `%-macro` names from all other reserved words in the HAL/S language.

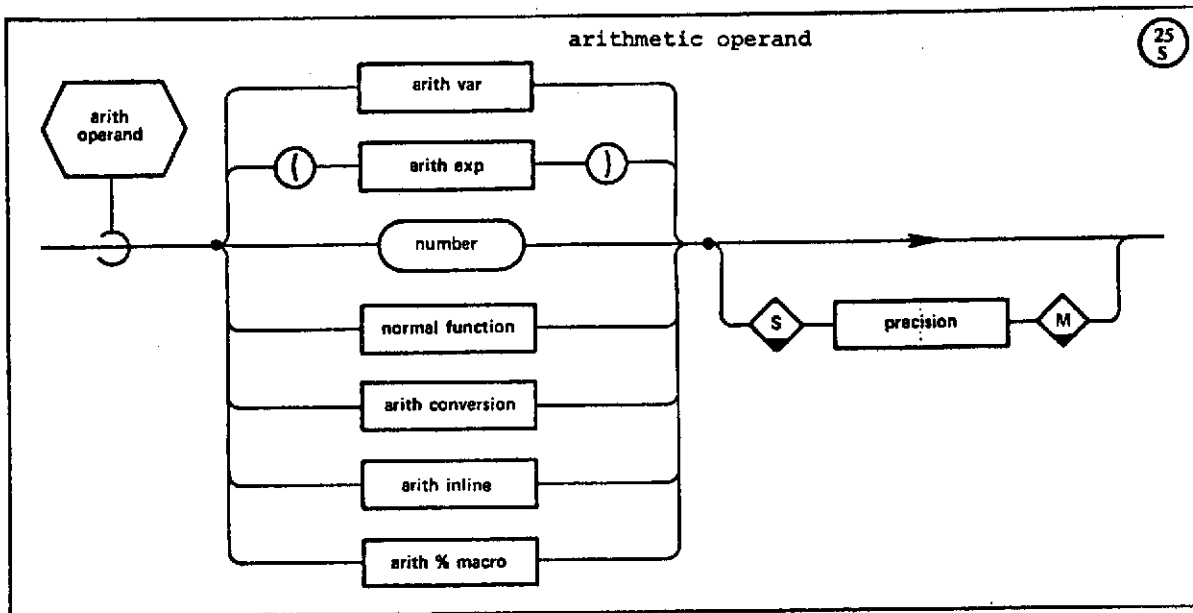
3. A series of one or more arguments of the %-macro reference may be supplied. The type, organization and number of the arguments supplied to the %-macro must be consistent with the requirements of the routine.
4. Details of <%-macro arg>s will be supplied with the definition of a given %-macro.

117

11.2.3 *Operand Reference Invocations*

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

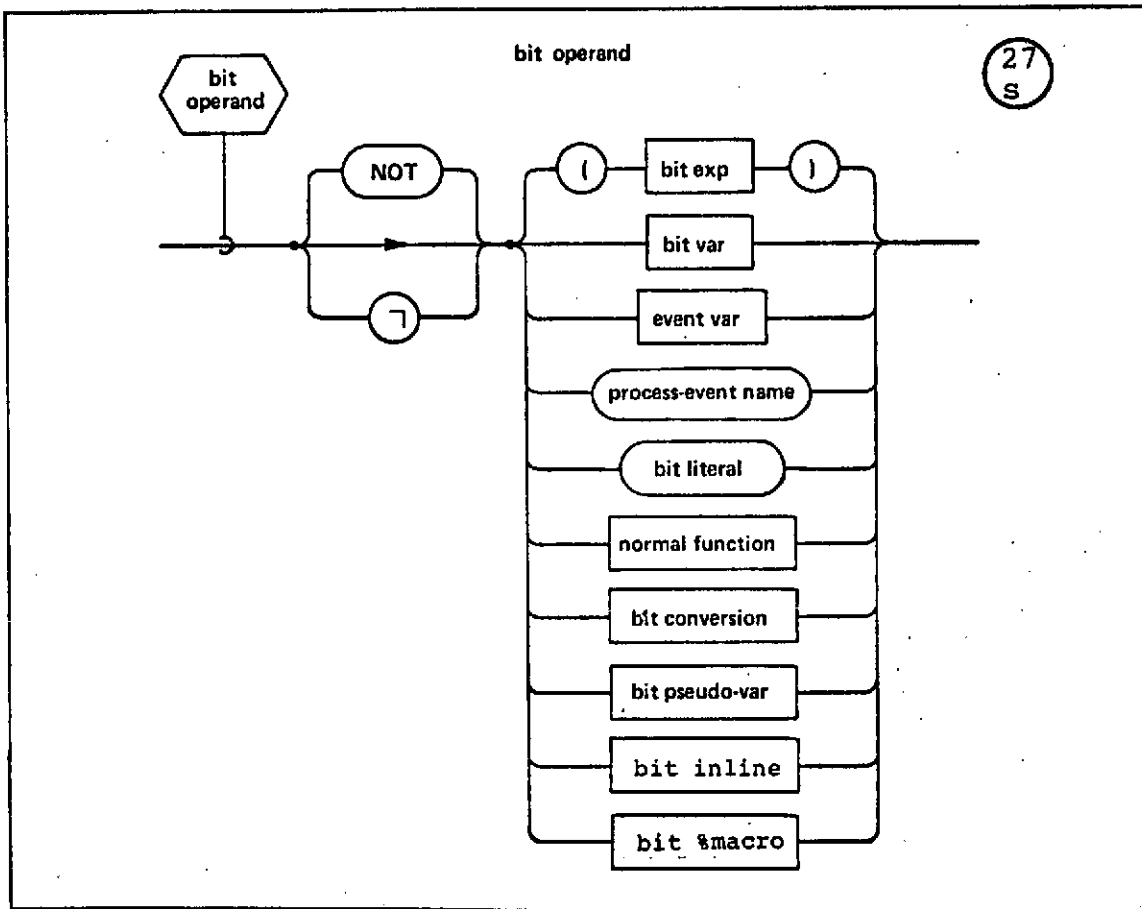
SYNTAX OF ARITHMETIC OPERAND:



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. <arith inline> is an inline function block which has an arithmetic <type spec> in its header statement.
3. <arith %-macro> is a reference to a %-macro which returns an arithmetic value (See 11.2.2 above).

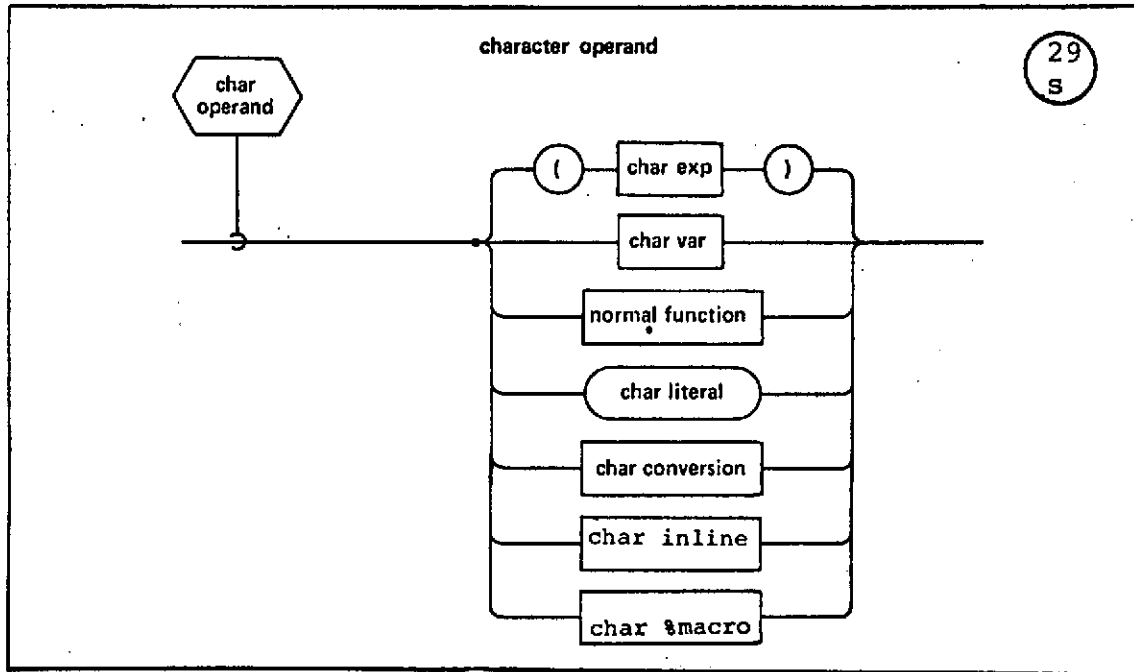
SYNTAX OF BIT OPERAND:



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. <bit inline> is an inline function block which has a bit string (BOOLEAN or BIT) <type spec> in its header statement.
3. <bit %macro> is a reference to a %-macro which returns a value of the BIT or BOOLEAN data types.

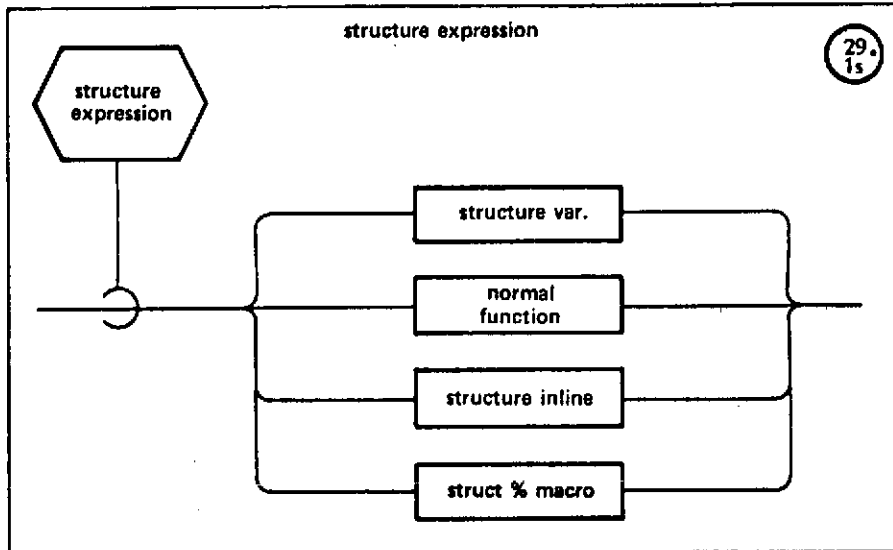
SYNTAX OF CHARACTER OPERAND:



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. <char inline> is an inline function block which has a CHARACTER <type spec> in its header statement.
3. <char %macro> is a reference to a %-macro which returns a value of the CHARACTER data type.

SYNTAX OF STRUCTURE EXPRESSION:



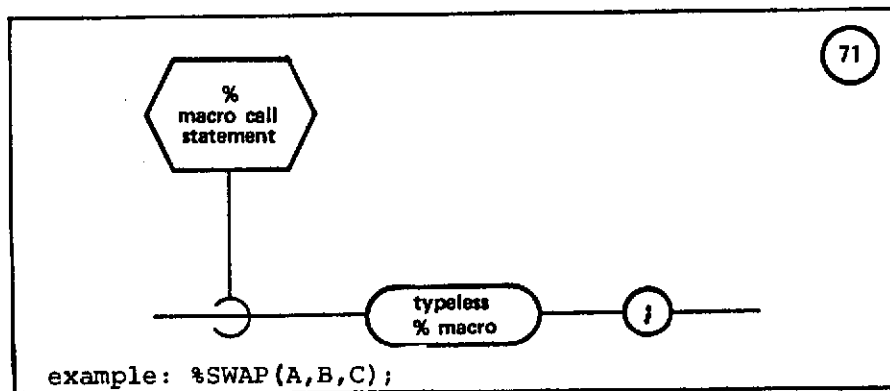
SEMANTIC RULES:

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. `<struct inline>` is an inline function block which has a structure `<type spec>` in its header statement.
3. `<struct %macro>` is a reference to a `%macro` which returns a value of a structure data type.

11.2.4 The %-Macro Call Statement

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

SYNTAX



SEMANTIC RULES:

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

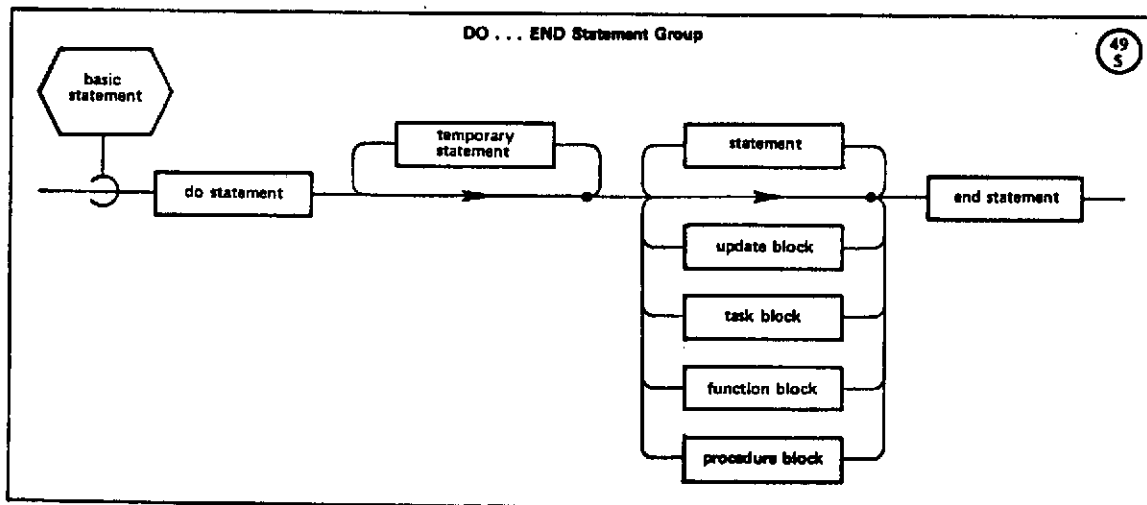
11.3 Temporary Variables

The extension of HAL/S data concepts to include a TEMPORARY variable form for use within DO groups is defined within the systems language facilities. The object of incorporating the TEMPORARY variable is to increase the optimization and efficiency of the object code produced by the compiler. Depending upon the details of the object machine, a temporary variable might be stored in a CPU register or a high speed, 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:

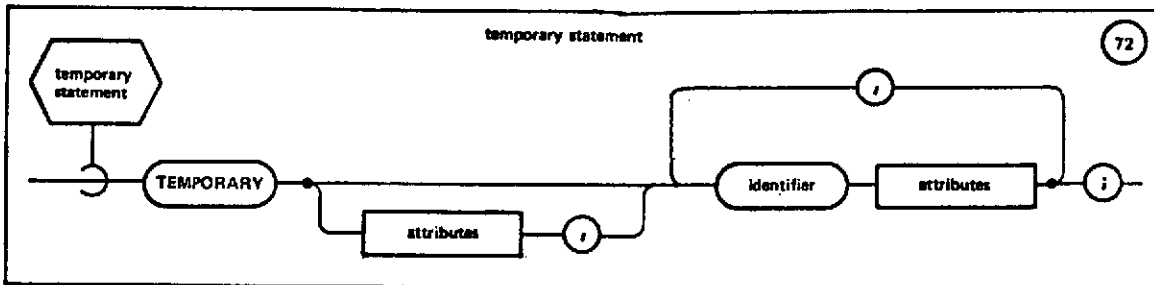


SEMANTIC RULE:

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:



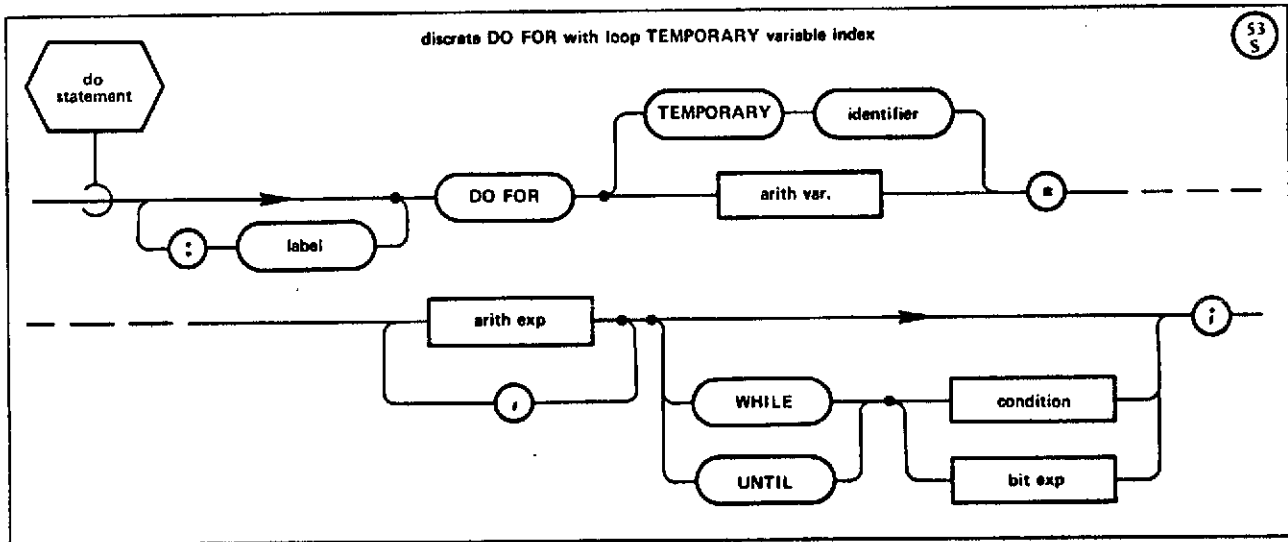
SEMANTIC RULES:

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

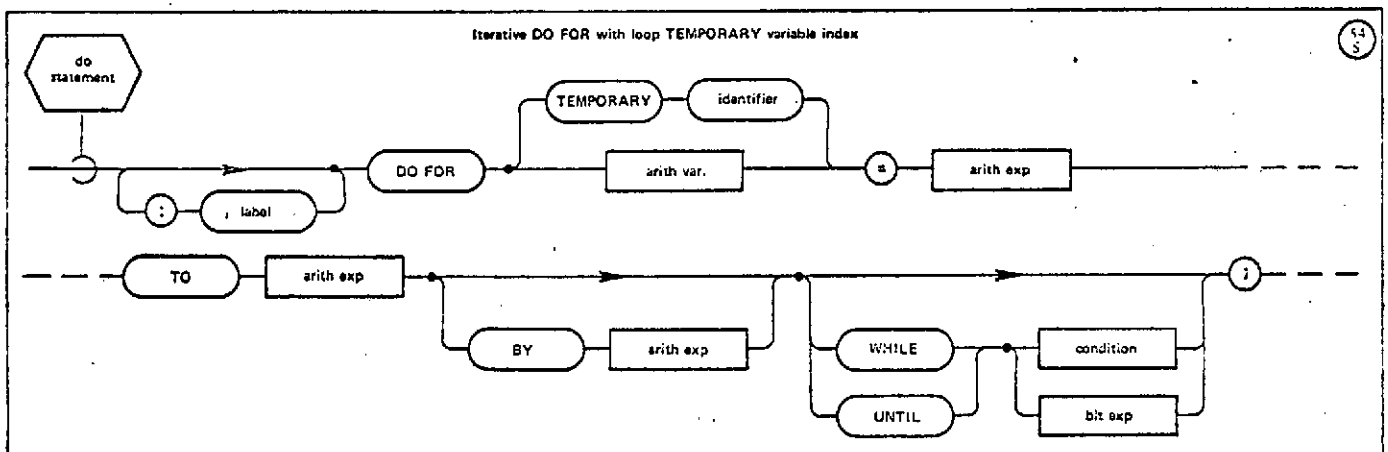
11.3.2 Loop TEMPORARY Variables

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.

II. 4 The NAME Facility

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

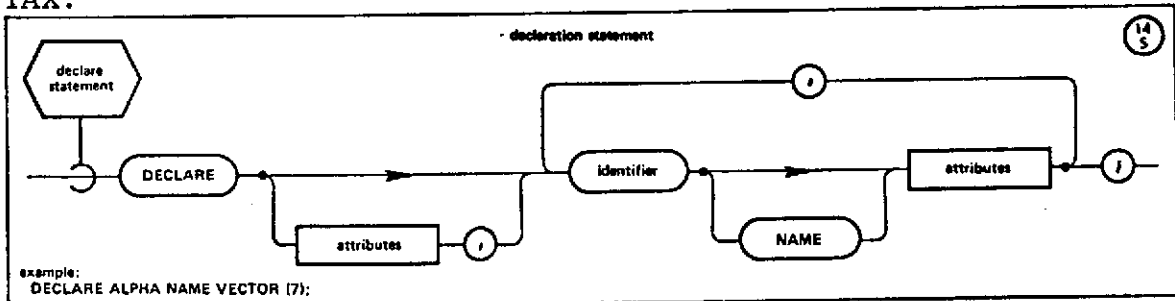
11.4.1 Identifiers with the NAME Attribute

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

INTEGER	PROGRAM
SCALAR	TASK
VECTOR	
MATRIX	
BIT	
BOOLEAN	
CHARACTER	
EVENT	
STRUCTURE	

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:



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/S 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 Chapter 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
```

SEMANTIC RULES (Data NAME Identifiers):

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

examples of data NAME variables

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

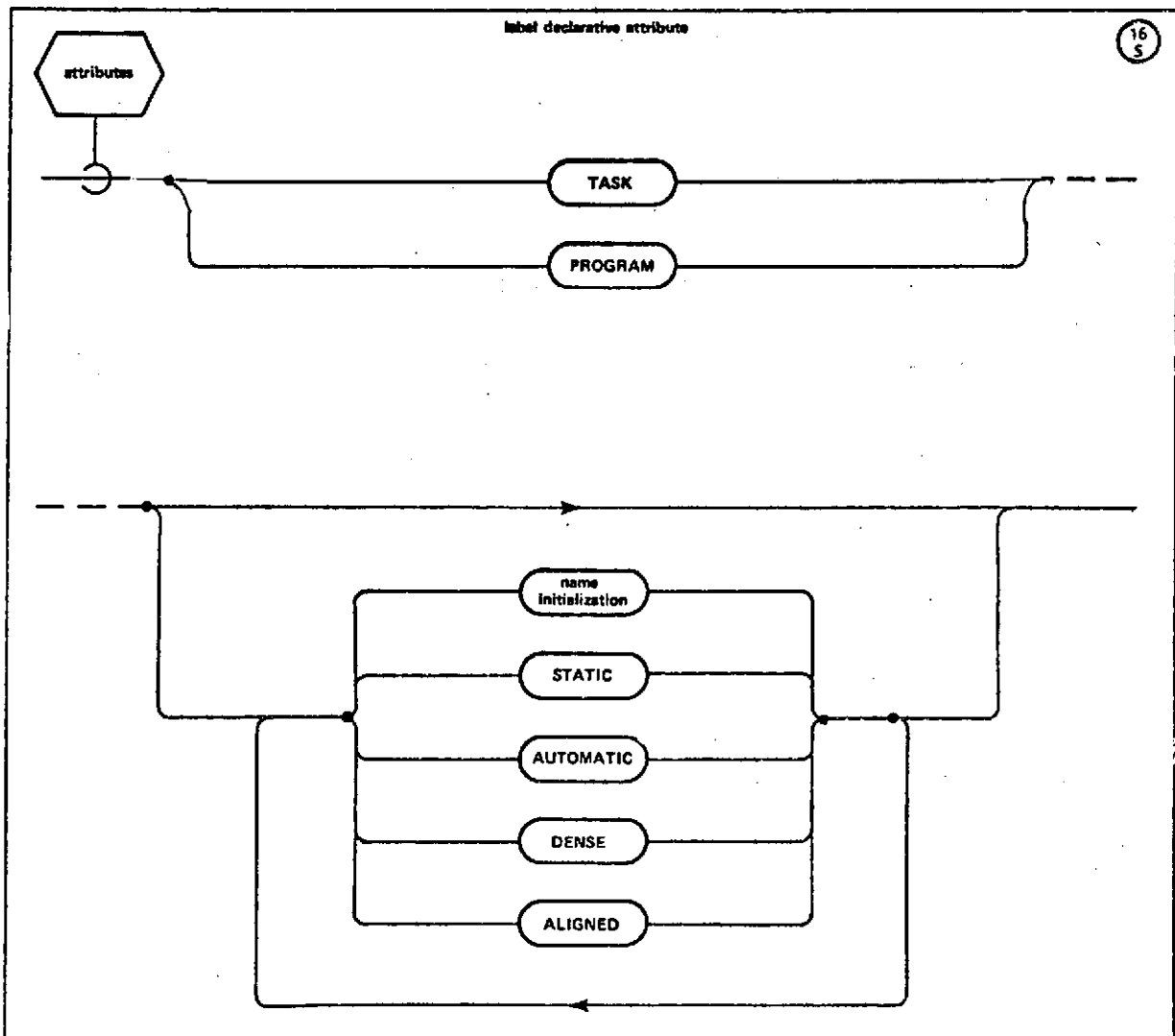
Z may point to X but not Y

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

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

126

SYNTAX:



126

SEMANTIC RULES (Label NAME Identifiers):

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

2. The following rules apply to NAME <identifiers> of the PROGRAM and TASK types:
 - The NAME <identifier> of a PROGRAM or TASK type always points to a PROGRAM or TASK block, respectively. A corollary of this rule is that <process event>s are never referenced by NAME identifiers of the PROGRAM or TASK types.

 - The only form of PROGRAM label declarations allowed are those with the NAME attribute.

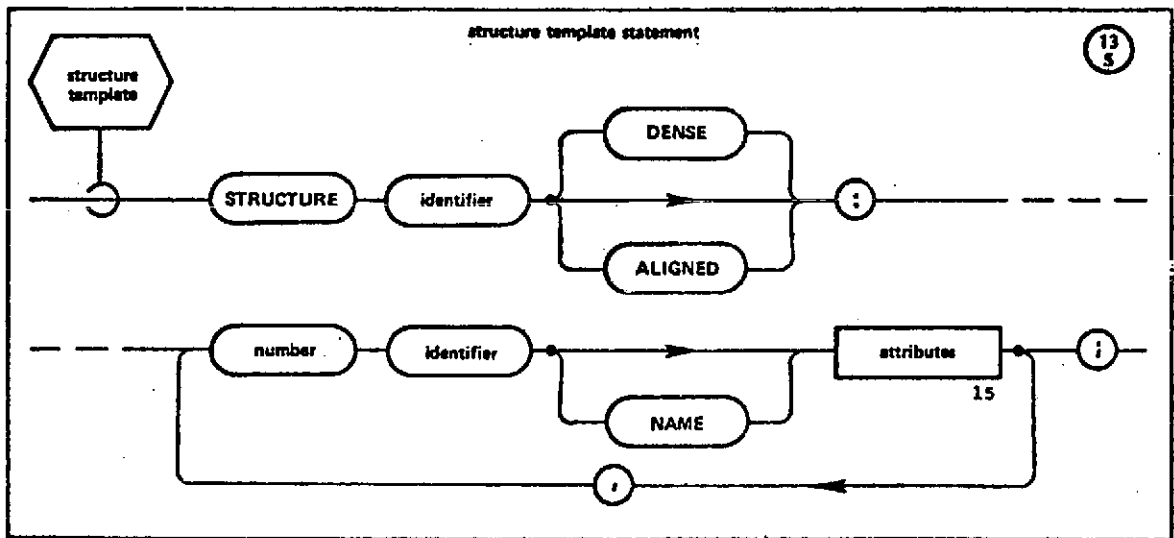
 - The program NAME <identifier> must always point to an external PROGRAM block name; therefore a block template is required for each PROGRAM which may be referenced by a NAME value.

THIS PAGE INTENTIONALLY LEFT BLANK

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 program, task, procedure & 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.
4. The REMOTE attribute may be applied to a structure terminal with the NAME attribute unless it is of EVENT type.

SEMANTIC RULES: Nested Structure Template References

1. Nested structure template references are special instances of structure terminals. 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:
 - Specification of multiple copies is still **not permitted**.
 - The reference may be to the structure template being defined (and of which the reference is a part). The implications of this are discussed later.

examples of structure NAME identifiers:

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

```
DECLARE P A-STRUCTURE;  
DECLARE PP NAME A-STRUCTURE;
```

P.Z is a NAME identifier which may point to
P.Y

PP is a NAME identifier which may point to
P

PP.Z is a NAME identifier which may point to
P.Z which is itself a NAME identifier
pointing somewhere. This is an instance
of double indirection.

P.ALPHA is a NAME identifier of A-structure type.
The consequences of this are discussed later.

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

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

examples of dereferenced NAME variables

```

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

```

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

TERMINATE P;	Means terminate Q.
Y = Y*X;	Puts the cross product of X with X in X.
Y ₁ = Y ₃ ;	Puts the third element of X into the first element.

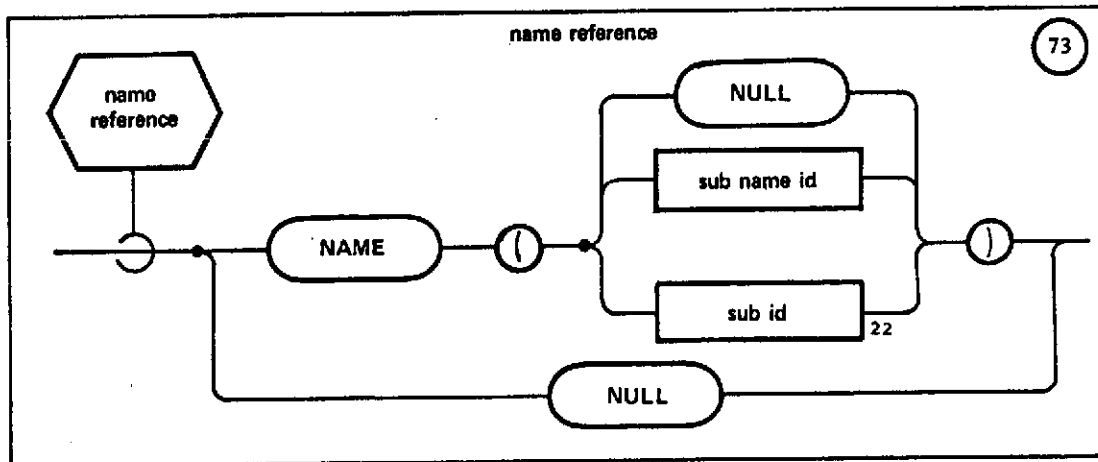
A special construct to be described in Sections 11.4.5 and 11.4.6 is required to reference or change the value of a NAME identifier (as opposed to referencing or changing the value to which it points).

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/S 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. <sub id> is any ordinary identifier, except an input parameter, a minor structure, an identifier declared with CONSTANT initialization, or an ACCESS-controlled identifier to which assignment access is "denied" or not asked for. <sub name id> is any NAME identifier.
2. Either of the above forms may possibly be modified by subscripting legal for its type and organization. Note, however, the following specific exceptions:

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

example:

```
DECLARE V NAME ARRAY(3) VECTOR;
```

```
:
```

```
NAME(V*:,1)
```

is illegal since it
violates the second excep-
tion of semantic rule 3 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 ACCESS control will compile without error only if implementation dependent ACCESS requirements for <sub id> are satisfied.
5. If <sub id> is unsubscripted, the construct delivers the location of the ordinary identifier specified. If it is subscripted, the construct delivers the location of the part of the specified identifier as determined by the form of the subscript. Subscripting can change the type and dimensions of <sub id> for matching purposes.
6. If <sub name id> is unsubscripted, the construct delivers the value of the NAME identifier specified. If it is subscripted, the value of the NAME identifier is taken to be the location of an ordinary identifier of compatible attributes, and the subscripting accordingly modifies the location delivered by the construct.

7. The two equivalent forms NULL and NAME(NULL) specify null NAME values.

examples:

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

NAME(X) yields the location of X.

NAME(NX) yields the value of NX (i.e. the location pointed to by NX).

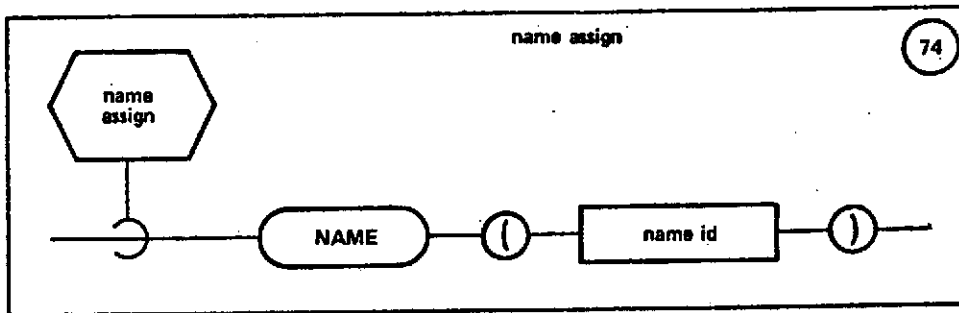
NAME(V₂) yields the location of the second element of V.

NAME(NV₃) 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:

SYNTAX:



SEMANTIC RULE:

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

example:

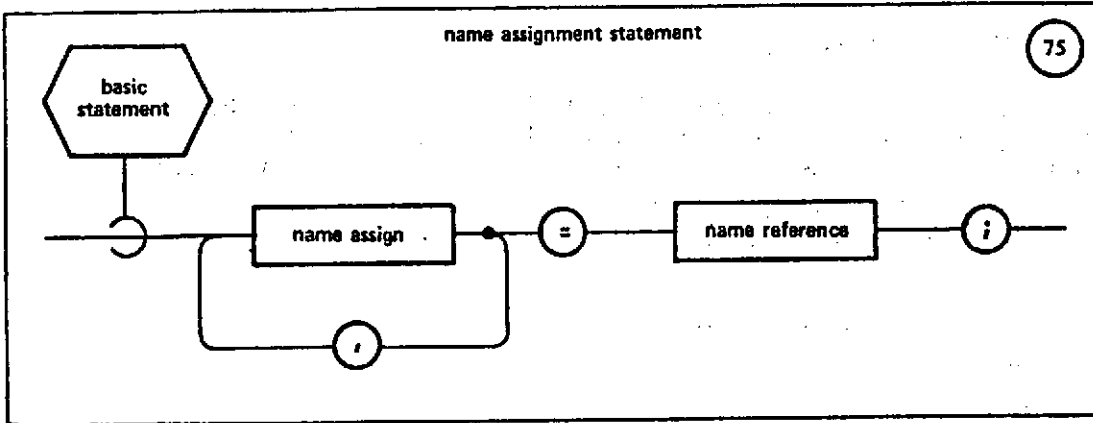
```
DECLARE X NAME MATRIX;
```

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

11.4.7 NAME Assignment Statements

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

SYNTAX :



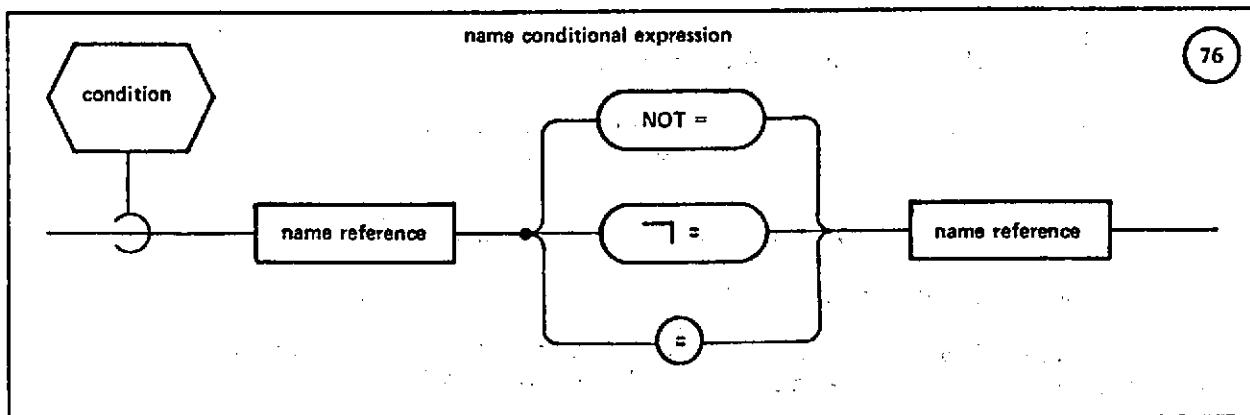
SEMANTIC RULES:

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

11.4.8 NAME Value Comparisons

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

SYNTAX :



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 contains a null value
                           (points at no location) then
                           return.

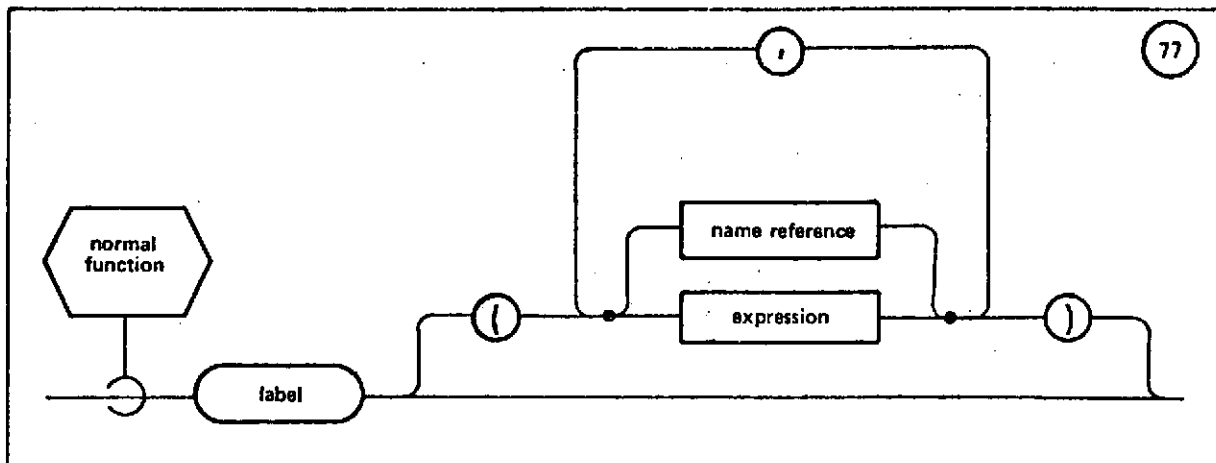
```

E

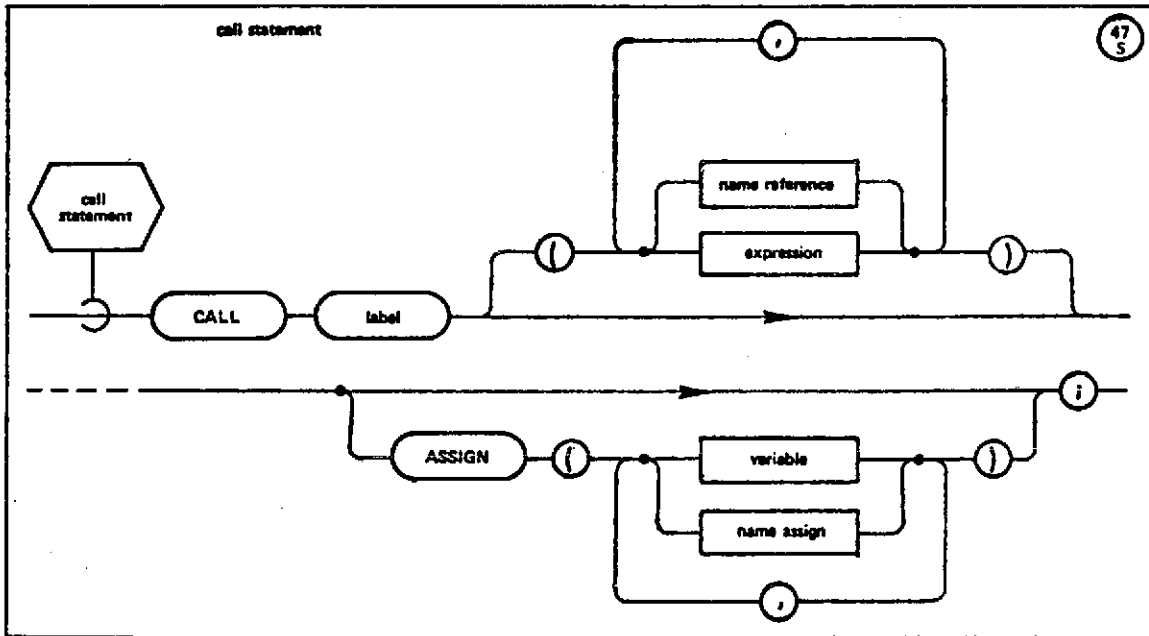
11.4.9 Argument Passage Considerations

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:



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:

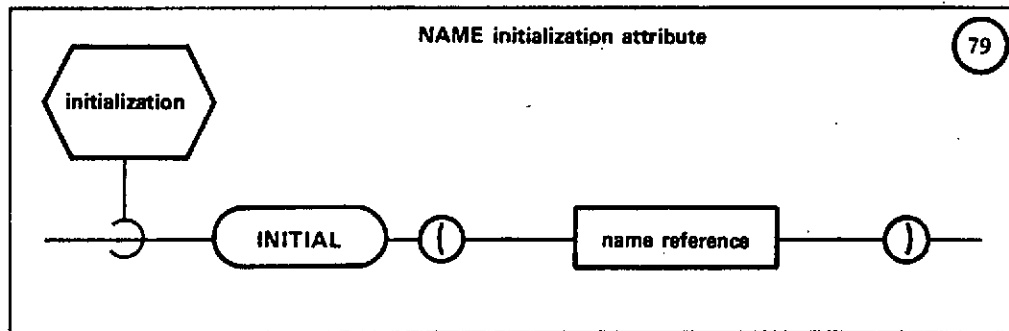
```
DECLARE XI SCALAR,  
        X2 NAME SCALAR;  
:  
P: PROCEDURE(A, B) ASSIGN(C, D);  
   DECLARE SCALAR, A NAME,  
           B,  
           C NAME,  
           D;  
   NAME(C) = NAME(A);  
   NAME(C) = NAME(B);  
:  
CLOSE;  
:  
NAME(X2) = NAME(X1);  
CALL P (NAME(X1), XI) ASSIGN(NAME(X2), XI);
```

illegal - B is an
input parameter

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

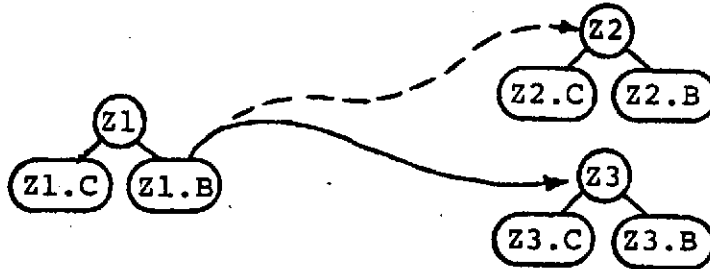
```
NAME(Z1.B) = NAME(Z2);
```

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.

Pictorially:

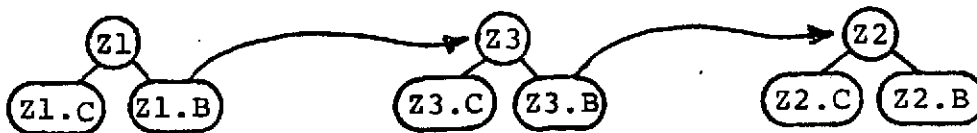


Now Z1.B.B is itself in turn a NAME identifier of A-structure type, so that if the NAME assignment

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

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

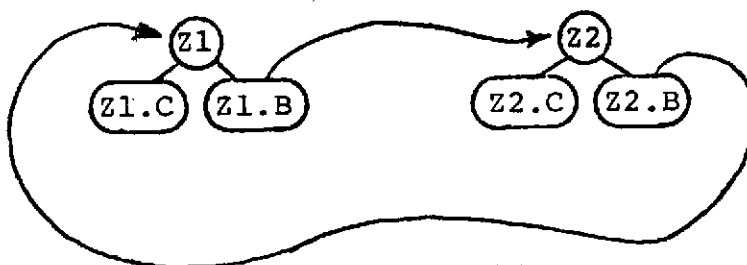
Pictorially:



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.

IMPLICATIONS OF SUBSCRIPTING STRUCTURE TERMINALS

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

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

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

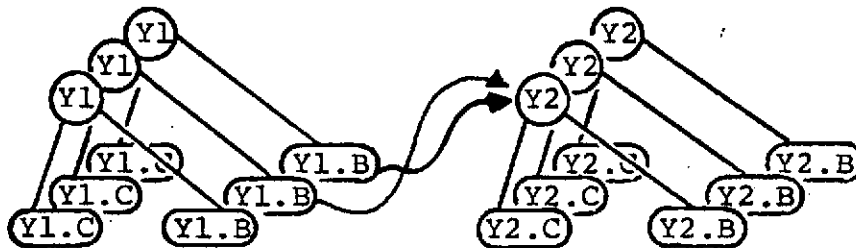
```
Y1.C2 AT 2;           (optional semicolon for clarity)
```

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.B2 AT 2) = NAME(Y22 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 Y2₂ 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.B2) = NAME(Y31);
```

Here the subscript 2 on the left-hand argument refers to copies of Y1 (this can be its only interpretation). Hence, by virtue of the fact that Y1.B₂ has previously been set up to point to Y2₁, this assignment causes Y2.B₁ to point to Y3₁.

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.

MANIPULATING STRUCTURES CONTAINING NAME TERMINALS

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

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

examples:

STRUCTURE IOTA:

1 LAMBDA NAME VECTOR,

1 KAPPA SCALAR;

DECLARE ALPHA IOTA-STRUCTURE(I0);

DECLARE BETA IOTA-STRUCTURE;

:

ALPHA₄ = BETA;

As a part of this assignment, the vector identifier (or NULL) pointed to by BETA.LAMBDA becomes the vector identifier pointed to by ALPHA.LAMBDA₄ 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 (ALPHA₅ 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 Operations, all NAME terminals are totally ignored. Name terminals can take part in FILE I/O. 113

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.

APPENDICES

11-40

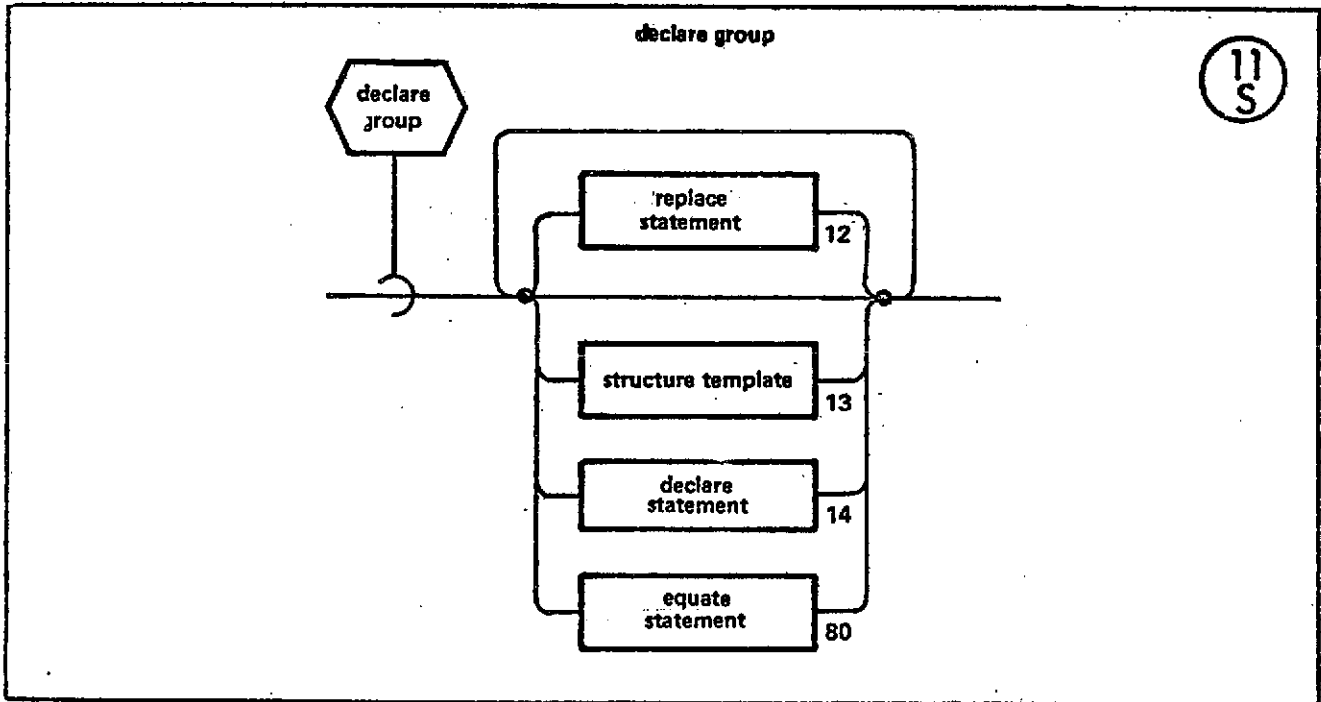
SEMANTIC RULES:

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

11.5.2 EQUATE Statement Placement

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

SYNTAX:



A. SYNTAX DIAGRAM SUMMARIES

A.1 SYNTAX PRIMITIVE REFERENCES

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

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

NOTES:

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

Syntactical Primitive	Diagram Number	Page
<arith var name>	[19]	[5-5]
	20	5-5
<argument>	12.1	4-6
<bit literal>	19	5-5
	20	5-5
<char literal>	[18]	[4-23]
	29	6-11
<char var name>	[18]	[4-20]
	19	5-5
	20	5-5
<event var name>	19	5-5
	20	5-5
<identifier>	8	3-15
	9	3-17
	12	4-4
	13	4-9
	14	4-12
	15	4-13
	14s	11-16
	13s	11-22
<label>	2	3-4
	3	3-6
	4	3-8
	5	3-10
	6	3-11
	10	3-19
	[18]	[4-23]
	38	6-23
	45	7-3
	46	7-5
	47	7-9

Syntactical Primitive	Diagram Number	Page
<label> (continued)	48	7-12
	50	7-15
	51	7-16
	52	7-17
	53	7-19
	54	7-21
	55	7-23
	56	7-24
	57	8-4
	58	8-9
	59	8-11
	60	8-12
	61	8-14
	62	8-15
	63	9-3
	64	9-7
	65	10-3
66	10-6	
68	10-10	
53s	11-14	
54s	11-14	
77	11-31	
47s	11-32	
<number>	13	4-9
	15	4-13
	16	4-18
	[18]	[4-23]
	25	6-6
	63	9-2
	64	9-3
	65	10-3
	66	10-6
	68	10-10
	16s	11-19
13s	11-22	
<process-event name>	27	6-8
	37	6-22
<template name>	17	4-19
<text>	12	4-4
	12.1	4-6

A.2 SYNTAX DIAGRAM CROSS REFERENCES

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

NOTES:

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

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<arith conversion>	39	6	6-27	25, 25s
<arith exp>	24	6	6-3	15, 17, 18, 22, 23, 25, 28, 32, 39, 51, 53, 54, 57, 60, 61, 67, 68, 25s, 54s
<arith operand>	25	6	6-6	24, 25s
<arith var>	19	5	5-5	20, 25, 53, 54, 25s, 54s
<array sub>	22	5	5-11	21
<arith inline>	25s	11	11-7	25
<arith % macro>	25s	11	11-7	25
<attributes>:				
data	15	4	4-13	
label	16	4	4-18	16s
name	16s	11	11-19	44, 45
<basic statement>:				
assignment	46	7	7-5	
name	75	11	11-27	47s
CALL	47	7	7-9	47s
name	47s	11	11-32	
CANCEL	58	8	8-9	
DO...END	49	7	7-14	
EXIT	56	7	7-24	
FILE	68	10	10-10	
GO TO	56	7	7-24	
name assign	74	11	11-27	
null	56	7	7-24	75, 76, 77, 47s
ON ERROR	63	9	9-3	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
READ	65	10	10-3	
READALL	65	10	10-3	
REPEAT	56	7	7-24	
RESET	62	8	8-15	
RETURN	48	7	7-12	
SCHEDULE	57	8	8-4	
SEND ERROR	64	9	9-7	
SET	62	8	8-15	
SIGNAL	62	8	8-15	
TERMINATE	59	8	8-11	
UPDATE PRIORITY	61	8	8-14	
WAIT	60	8	8-12	
WRITE	66	10	10-6	
<bit conversion>	40	6	6-31	27,27s
<bit exp>	26	6	6-7	23,27,33,41,45,52,53,54,27s,54s
<bit inline>	27s	11	11-8	27
<bit % macro>	27s	11	11-8	27
<bit operand>	27	6	6-8	26,27s
<bit pseudo-var>	42	6	6-35	20,27,27s
<bit var>	19	5	5-5	20,27,27s
<char conversion>	41	6	6-33	29,29s
<char exp>	28	6	6-10	23,29,34,30,29s
<char operand>	29	6	6-11	28,29s
<char var>	19	5	5-5	20,29,29s

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<char inline>	29s	11	11-9	29
<char % macro>	29s	11	11-9	29
<closing>	10	3	3-19	2,3,4,5,6,69
<comparison>:				31
arithmetic	32	6	6-15	
bit	33	6	6-17	
character	34	6	6-18	
structure	35	6	6-19	
<compilation>	1	3	3-2	
<component sub>	22	5	5-11	21
<compool block>	5	3	3-10	1
<compool header>	7	3	3-14	5,6
<compool template>	6	3	3-11	1
<condition>	30	6	6-13	31,45,52,53,54,54s
name	76	11	11-30	
<conditional operand>	31	6	6-14	30
<declare group>	11	4	4-3	2,3,4,5,6,69
<declare statement>	14	4	4-12	11,14s,11s
name	14s	11	11-16	
<do statement>:				49,49s
CASE	51	7	7-16	
discrete FOR	53	7	7-19	
temporary var	53s	11	11-14	
iterative FOR	54	7	7-21	

E

135

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
temporary var	54s	11	11-14	
simple	50	7	7-15	
UNTIL	52	7	7-16	
WHILE	52	7	7-17	
<end statement>	55	7	7-23	49,49s
<equate statement>	80	11	11-40	11s
<event exp>	36	6	6-21	37,57,60
<event operand>	37	6	6-22	36
<event var>	19	5	5-5	20,27,37,62
<expression>	23	6	6-2	18,38,39,40,41,42,46,47,48,66, 68,70
<file exp>	68	10	10-10	68
<function block>	3	3	3-6	1,2,3,4,49,49s
<function header>	9	3	3-17	3,6
<function template>	6	3	3-11	1
<inline function>	69	11	11-2	
<initial list>	18	4	4-20	18
<initialization>	18	4	4-23	15
name	79	11	11-33	16s
<i/o control>	67	10	10-8	65,66
<name>	14s	11	11-16	
<name reference>	75	11	11-30	
<normal function>	38	6	6-23	25,27,29,77
name	77	11	11-31	
<precision>	43	6	6-38	1,2,3,4,49,49s
<procedure block>	3	3	3-6	3,6

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<procedure header>	8	3	3-15	3,6
<procedure template>	6	3	3-11	1
<program block>	2	3	3-4	1
<program header>	7	3	3-14	2
<% macro>	70	11	11-5	25s
typeless % macro	71	11	11-11	
<radix>	Note 2.	6		40,41
<replace statement>	12	4	4-4	11,11s
parametric	12.1	4	4-6	
<statement>:				
basic	44	7	7-2	
IF	45	7	7-3	
temporary	72	11	11-13	
<structure exp>	29.1	6	6-12	
<structure sub>	22	5	5-11	21
<struct inline>	29.1s	11	11-10	29.1
<struct % macro>	29.1	11	11-10	29.1
<structure template>	13	4	4-9	11,11s
name	13s	11	11-22	
<structure var>	19	5	5-5	20,23,35,29,15,29.1
<sub exp>	22	5	5-11	22
<sub name id>	73	11	11-26	
<subscript>	21	5	5-8	19,39,40,41,42
<task block>	3	3	3-6	2,49,49s

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<task header>	7	3	3-14	3
<type spec>	17	4	4-19	9,15,16,69
<update block>	4	3	3-8	2,3,49,69,49s
<update header>	7	3	3-14	4
<variable>	20	5	5-5	42,46,47,65,68
<temporary statement>	49s 72	11 11	11-12 11-13	53s,54s

A.3 SYNTAX DIAGRAM LISTING *

DIAGRAM #	TITLE	PAGE
1	unit of compilation	3-2
2	PROGRAM block	3-4
3	PROCEDURE, FUNCTION and TASK blocks	3-6
4	UPDATE block	3-8
5	COMPOOL block	3-10
6	block templates: PROGRAM, PROCEDURE, FUNCTION and COMPOOL templates	3-11
7	simple header statement	3-14
8	PROCEDURE header statement	3-15
9	FUNCTION header statement	3-17
10	Closing of block	3-19
11	declare group	4-3
11s	EQUATE statement placement in declare group	11-42
12	REPLACE statement	4-4
13	structure template statement	4-9
13s	structure template statement/NAME attribute	11-22
14	declare statement	4-12
14s	declaration statement/NAME attribute	11-16
15	data declarative attributes	4-13
16	label declarative attributes	4-18
16s	label declarative attributes/PROGRAM-TASK	11-19
17	type specification	11-19
18	initialization specification	4-23
19	<var>: arithmetic, bit, character, structure, event variables	5-5
20	variable	5-5
21	subscript construct	5-8
22	component, array, and structure subscripts	5-11
23	expression	6-2
24	arithmetic expression	6-3
25	arithmetic operand	6-6
25s	arithmetic operand inline function block/ %-macros	11-7

*Note that an "s" suffix identifies a modified Systems Language diagram.

DIAGRAM #	TITLE	PAGE
26	bit expression	6-7
27	bit operand	6-8
27s	bit operand inline function block/ %macros	11-8
28	character expression	6-10
29	character operand	6-11
29s	character operand inline function block/ %macros	11-9
29.1	structure expression	6-12
29.1s	structure expression inline function block/ %macros	11-10
30	conditional expression	6-13
31	conditional operand	6-14
32	arithmetic comparison	6-15
33	bit comparison	6-17
34	character comparison	6-18
35	structure comparison	6-19
36	event expression	6-21
37	event operand	6-22
38	normal function	6-23
39	arithmetic conversion function	6-27
40	bit conversion function	6-31
41	character conversion function	6-33
42	SUBBIT pseudo-variable	6-35
43	precision specifier	6-38
44	basic statement	7-2
45	IF statement	7-3
46	assignment statement	7-5
47	CALL statement	7-9
47s	CALL statement with NAME	11-32
48	RETURN statement	7-12
49	DO...END statement group	7-14
49s	DO...END statement group/temporary variable	11-12
50	simple DO statement	7-15
51	DO CASE statement	7-16
52	DO WHILE and UNTIL statements	7-17
53	discrete DO FOR statement	7-18
53s	discrete DO FOR statement/temporary variable	11-14
54	iterative DO FOR statement	7-21
54s	iterative DO FOR statement/temporary variable	11-14
55	END statement	7-23

DIAGRAM #	TITLE	PAGE
56	other basic statements: GO TO, "null", EXIT and REPEAT statements	7-24
57	SCHEDULE statement	8-4
58	CANCEL statement	8-9
59	TERMINATE statement	8-11
60	WAIT statement	8-12
61	UPDATE PRIORITY statement	8-14
62	SET, SIGNAL, and RESET statement	8-15
63	ON ERROR statement	9-3
64	SEND ERROR statement	9-7
65	READ and READALL statements	10-3
66	WRITE statement	10-6
67	i/o control function	10-8
68	FILE statements	10-10
69	inline function block	11-2
70	%-macro statement	11-5
71	%-macro call	11-11
72	temporary statement	11-13
73	NAME reference	11-26
74	NAME assign	11-29
75	NAME assignment statement	11-30
76	NAME conditional expression	11-30
77	normal function reference	11-31
79	NAME initialization attribute	11-33
80	EQUATE statement	11-40

B. HAL/S KEYWORDS

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

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

C. BUILT-IN FUNCTIONS

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

ARITHMETIC FUNCTIONS									
<ul style="list-style-type: none"> ● arguments may be INTEGER or SCALAR types ● in functions with one argument, result type matches argument type (except as specifically noted) ● in functions with two arguments, result type is scalar if either or both arguments are scalar; otherwise the result type is integer ● arrayed arguments cause multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match 									
Name, Arguments	Comments								
ABS(α)	$ \alpha $								
CEILING(α)	smallest integer $\geq \alpha$								
DIV(α, β)	integer division α/β (arguments rounded to integers)								
FLOOR(α)	largest integer $\leq \alpha$								
MOD(α, β)	$\alpha \text{ MOD } \beta$								
ODD(α)	<table style="display: inline-table; border: none;"> <tr> <td style="padding-right: 10px;">TRUE</td> <td style="padding-right: 10px;">1</td> <td style="padding-right: 10px;">if α odd</td> <td rowspan="2" style="font-size: 2em; padding: 0 10px;">}</td> <td rowspan="2" style="vertical-align: middle;">result is Boolean</td> </tr> <tr> <td>FALSE</td> <td>0</td> <td>if α even</td> </tr> </table>	TRUE	1	if α odd	}	result is Boolean	FALSE	0	if α even
TRUE	1	if α odd	}	result is Boolean					
FALSE	0	if α even							
REMAINDER(α, β)	signed remainder of integer division α/β (argument rounded to integer)								
ROUND(α)	nearest integral value to α								
SIGN(α)	<table style="display: inline-table; border: none;"> <tr> <td style="padding-right: 10px;">+1</td> <td style="padding-right: 10px;">$\alpha > 0$</td> </tr> <tr> <td>-1</td> <td>$\alpha < 0$</td> </tr> </table>	+1	$\alpha > 0$	-1	$\alpha < 0$				
+1	$\alpha > 0$								
-1	$\alpha < 0$								
SIGNUM(α)	<table style="display: inline-table; border: none;"> <tr> <td style="padding-right: 10px;">+1</td> <td style="padding-right: 10px;">$\alpha > 0$</td> </tr> <tr> <td>0</td> <td>$\alpha = 0$</td> </tr> <tr> <td>-1</td> <td>$\alpha < 0$</td> </tr> </table>	+1	$\alpha > 0$	0	$\alpha = 0$	-1	$\alpha < 0$		
+1	$\alpha > 0$								
0	$\alpha = 0$								
-1	$\alpha < 0$								
TRUNCATE(α)	largest integer $\leq \alpha $ times SIGNUM (integer $\overline{\alpha}$)								

ALGEBRAIC FUNCTIONS	
<ul style="list-style-type: none"> arguments may be integer or scalar types - conversion to scalar occurs with integer arguments result type is always scalar arrayed arguments cause multiple invocations of the function, one for each array element angular values are supplied or delivered in radians. 	
Name, Arguments	Comments
ARCCOS (α)	$\cos^{-1} \alpha \quad \alpha \leq 1$
ARCCOSH (α)	$\cosh^{-1} \alpha \quad \alpha \geq 1$
ARCSIN (α)	$\sin^{-1} \alpha, \quad \alpha \leq 1$
ARCSINH (α)	$\sinh^{-1} \alpha$
ARCTAN2 (α, β)	$-\pi < \tan^{-1}(\alpha/\beta) \leq \pi$ Proper Quadrant if: $\left. \begin{array}{l} \alpha = k \sin \theta \\ \beta = k \cos \theta \end{array} \right\} k > 0$
ARCTAN (α)	$\tan^{-1} \alpha$
ARCTANH (α)	$\tanh^{-1} \alpha \quad \alpha < 1$
COS (α)	$\cos \alpha$
COSH (α)	$\cosh \alpha$
EXP (α)	e^{α}
LOG (α)	$\log_e \alpha, \quad \alpha > 0$
SIN (α)	$\sin \alpha$
SINH (α)	$\sinh \alpha$
SQRT (α)	$\sqrt{\alpha}, \quad \alpha \geq 0$
TAN (α)	$\tan \alpha$
TANH (α)	$\tanh \alpha$

102

VECTOR-MATRIX FUNCTIONS

- arguments are vector or matrix types as indicated
- result types are as implied by mathematical operation
- arrayed arguments cause multiple invocations of the function, one for each array element

Name, Arguments	Comments
ABVAL(α)	length of vector α
DET(α)	determinant of square matrix α
INVERSE(α)	inverse of nonsingular square matrix α
TRACE(α)	sum of diagonal elements of square matrix α
TRANSPOSE(α)	transpose of matrix α
UNIT(α)	unit vector in same direction as vector α

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

116

CHARACTER FUNCTIONS

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

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

ARRAY FUNCTIONS

- arguments are n-dimensional arrays where n is arbitrary
- arguments are integer or scalar type
- result type matches argument type and is unarrayed

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

CHARACTER FUNCTIONS

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

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

BIT FUNCTIONS		
<ul style="list-style-type: none"> ● arguments are bit type ● result is bit type ● arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match 		
Name, Arguments	Result Type	Comments
XOR(α, β)	bit	Result is Exclusive OR of α and β . Length of result is length of longer argument. Shorter argument is left padded with binary zeros to length of longer argument.

ARRAY FUNCTIONS	
<ul style="list-style-type: none"> ● arguments are n-dimensional arrays where n is arbitrary ● arguments are integer or scalar type ● result type matches argument type and is unarrayed 	
Name, Parameters	Comments
MAX(α)	maximum of all elements of α
MIN(α)	minimum of all elements of α
PROD(α)	product of all elements of α
SUM(α)	sum of all elements of α

D. STANDARD CONVERSION FORMATS

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

CONVERSIONS TO INTEGER TYPE:

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

137

CONVERSIONS TO SCALAR TYPE:

- An integer type is converted directly to scalar form. Depending on the implementation, and the precisions, some decimal places of accuracy may be lost during conversion.
- A bit type is converted to scalar type by first converting it to double precision integer type according to the rule previously given, and then applying the integer to scalar conversion.
- A character type is convertible to integer type only if its value represents a legal scalar- or integer-valued literal (e.g. '-1.5E-7'). See Section 2.3.3 for details of arithmetic literals. Other values cause error conditions to arise.

137

CONVERSIONS TO BIT TYPE:

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

CONVERSIONS TO CHARACTER TYPE:

- An integer type is converted to the representation

```

      dddd      (positive)
     -dddd      (negative)

```

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

- A scalar type is converted to the representation

```

  /d.ddddE±dd      (positive)
 -d.ddddE±dd      (negative)

```

(except scalar 0 is converted to 0.0).

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

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

E. STANDARD EXTERNAL FORMATS

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

OUTPUT FORMATS

1. Integer Type:

- The value of an integer is represented by a string of decimal digits, preceded if it is negative by a - sign. Leading zeroes are suppressed.
- The string of digits is right justified in a field of fixed width. The width depends on the implementation, and on the precision of the integer.

2. Scalar Type:

- If the value of a scalar is positive it is represented by

$\text{ } \backslash d . d d d d d d d E \pm d d$

where d represents a decimal digit. One non-zero digit appears before the decimal point. The numbers of digits in the fractional part and exponent are fixed, and depend on the implementation and the precision of the scalar. Leading zeroes in the exponent are not suppressed. The representation includes a leading blank (\backslash).

- A negative value has the same form except that a - sign precedes the first decimal digit.
- If the value is exactly zero, it is represented as 0.0.
- The representation of a scalar is contained in a field of fixed width. The width is dependent on the implementation and the precision of the scalar. Justification is such that the decimal point occupies a fixed, precision dependent position in the field.

3. Bit Type (including BOOLEAN):

- There are two different representations of values of bit variables.
- The first representation consists of a string of binary digits corresponding to the bit variable. Leading binary zeros are not suppressed. The field width is equal to the number of binary digits in the string plus an inserted blank following every fourth digit (to enhance readability). This form is not compatible with the READ input (see Section 10.1.1).
- In the alternate representation, the string of binary digits plus inserted blanks is enclosed in the apostrophes. The field width is equal to the total of the number of digits, blanks and two apostrophes.

4. Character Type:

- There are two different representations of values of character variables.
- The first representation merely consists of the string of characters comprising the value. The field width is equal to the number of characters in the string. This representation is not compatible with READ input (see Section 10.1.1).
- In the alternate representation, the string of characters is enclosed in apostrophes, and all internal apostrophes are converted to apostrophe pairs. The field width is equal to the total number of characters in the string, including added apostrophes.

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

INPUT FORMATS

1. Scalar and Integer Types:

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

- The floating-point representation is either

ddd.dddd

or dddd.dddd $\left\{ \begin{array}{c} E \\ B \\ H \end{array} \right\} \pm dd$

where d is a decimal digit. Any number of digits is allowed in the mantissa to an implementation dependent maximum. The decimal point may appear in any position. E, B, and H represent the exponent digits to be powers of 10, 2 and 16 respectively. A choice of one is indicated. The maximum number of digits in the exponent is implementation dependent. For bit and integer types, the representation is rounded to the nearest integral value. For bit types the binary representation of the result is taken.

- The floating-point representation may be prefixed by + or - signs to indicate the sign of the value. Without such prefix the value is positive.

2. Character Type:

- The representation of character type is a string of characters from the HAL/S extended set enclosed in apostrophes. The number of characters may vary between zero (a "null string") and an implementation dependent maximum. Within the string apostrophes must be represented by an apostrophe pair.

3. Bit Type:

- The representation of bit type is a string of '1's and '0's enclosed in apostrophes. Imbedded blanks are ignored. The number of digits may vary between one and an implementation maximum.

F. COMPILE-TIME COMPUTATIONS

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

- declaration of dimensions;
- initialization;
- subscripting.

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

ARITHMETIC EXPRESSIONS (see Section 6.1.1)

1. <arith exp>s of integer and scalar type only can be computable at compile time.
2. The operators of such <arith exp>s are limited to:
 - +
 -
 - <> (multiply)
 - /
 - **
3. The <arith operand>s of such <arith exp>s may either be <number>s or unarrayed unsubscripted simple variables¹ of integer or scalar type. Such variables must previously have been declared, and initialized using the CONSTANT form.

¹ see Section 4.5

4. The following built-in functions are also legal:

SIN	EXP	DATE
COS	LOG	CLOCKTIME
TAN	SQRT	

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

BIT EXPRESSIONS (see Section 6.1.2)

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

~
&
|

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

CHARACTER EXPRESSIONS (see Section 6.1.3)

1. The catenation operator (||) only may appear in <char exp>s computable at compile time.

2. The <char operand>s of such <char exp>s must be either <char literal>s, <arith exp>s computable at compile time, or unarrayed unsubscripted simple variables of character type. Such variables must previously have been declared, and initialized using the CONSTANT form.

In some implementations, additional forms may also be computed at compile time. They will not, however, be regarded as legal in contexts where compile time computability is enforced semantically.

G. WORKING GRAMMAR

E

Edited Grammar

```

1  <COMPILATION> ::= <COMPILE LIST> _|_
2  <COMPILE LIST> ::= <BLOCK DEFINITION>
3                      | <COMPILE LIST> <BLOCK DEFINITION>
4  <ARITH EXP> ::= <TERM>
5                      | + <TERM>
6                      | - <TERM>
7                      | <ARITH EXP> + <TERM>
8                      | <ARITH EXP> - <TERM>
9  <TERM> ::= <PRODUCT>
10                      | <PRODUCT> / <TERM>
11 <PRODUCT> ::= <FACTOR>
12                      | <FACTOR> * <PRODUCT>
13                      | <FACTOR> . <PRODUCT>
14                      | <FACTOR> <PRODUCT>
15 <FACTOR> ::= <PRIMARY>
16                      | <PRIMARY> <*> <FACTOR>
17 <*> ::= **
18 <PRE PRIMARY> ::= ( <ARITH EXP> )
19                      | <NUMBER>
20                      | <COMPOUND NUMBER>
21 <ARITH FUNC HEAD> ::= <ARITH FUNC>
22                      | <ARITH CONV> <SUBSCRIPT>
23 <ARITH CONV> ::= INTEGER
24                      | SCALAR
25                      | VECTOR
26                      | MATRIX
27 <PRIMARY> ::= <ARITH VAR>
28 <PRE PRIMARY> ::= <ARITH FUNC HEAD> ( <CALL LIST> )
29 <PRIMARY> ::= <MODIFIED ARITH FUNC>
30                      | <ARITH INLINE DEF> <BLOCK BODY> <CLOSING> ;
31                      | <PFE PRIMARY>
32                      | <PFE PRIMARY> <QUALIFIER>
33 <OTHER STATEMENT> ::= <ON PHRASE> <STATEMENT>
34                      | <IF STATEMENT>
35                      | <LABEL DEFINITION> <OTHER STATEMENT>

```

```

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

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

40 <BASIC STATEMENT> ::= <LABEL DEFINITION> <BASIC STATEMENT>
41 | <ASSIGNMENT> ;
42 | EXIT ;
43 | EXIT <LABEL> ;
44 | REPEAT ;
45 | REPEAT <LABEL> ;
46 | GO TO <LABEL> ;
47 | ;
48 | <CALL KEY> ;
49 | <CALL KEY> ( <CALL LIST> ) ;
50 | <CALL KEY> <ASSIGN> ( <CALL ASSIGN LIST> ) ;
51 | <CALL KEY> ( <CALL LIST> ) <ASSIGN> ( <CALL ASSIGN LIST> ) ;
52 | RETURN ;
53 | RETURN <EXPRESSION> ;
54 | <DO GROUP HEAD> <ENDING> ;
55 | <READ KEY> ;
56 | <READ PHRASE> ;
57 | <WRITE KEY> ;
58 | <WRITE PHRASE> ;
59 | <FILE EXP> = <EXPRESSION> ;
60 | <VARIABLE> = <FILE EXP> ;
61 | <WAIT KEY> FOR DEPENDENT ;
62 | <WAIT KEY> <ARITH EXP> ;
63 | <WAIT KEY> UNTIL <ARITH EXP> ;
64 | <WAIT KEY> FOR <BIT EXP> ;
65 | <TERMINATOR> ;
66 | <TERMINATOR> <TERMINATE LIST> ;
67 | UPDATE PRIORITY TO <ARITH EXP> ;
68 | UPDATE PRIORITY <LABEL VAR> TO <ARITH EXP> ;
69 | <SCHEDULE PHRASE> ;
70 | <SCHEDULE PHRASE> <SCHEDULE CONTROL> ;
71 | <SIGNAL CLAUSE> ;
72 | SEND ERROR <SUBSCRIPT> ;
73 | <ON CLAUSE> ;
74 | <ON CLAUSE> AND <SIGNAL CLAUSE> ;
75 | OFF ERROR <SUBSCRIPT> ;
76 | <% MACRO NAME> ;
77 | <% MACRO HEAD> <% MACRO ARG> ) ;

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

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

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

```

```

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

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

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

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

108 <COMPARISON> ::= <ARITH EXP> <RELATIONAL OP> <ARITH EXP>
109                 | <CHAR EXP> <RELATIONAL OP> <CHAR EXP>
110                 | <BIT CAT> <RELATIONAL OP> <BIT CAT>
111                 | <STRUCTURE EXP> <RELATIONAL OP> <STRUCTURE EXP>
112                 | <NAME EXP> <RELATIONAL OP> <NAME EXP>

113 <RELATIONAL FACTOR> ::= <REL PRIM>
114                   | <RELATIONAL FACTOR> <AND> <REL PRIM>

115 <RELATIONAL EXP> ::= <RELATIONAL FACTOR>
116                   | <RELATIONAL EXP> <OR> <RELATIONAL FACTOR>

117 <REL PRIM> ::= ( <RELATIONAL EXP> )
118           | <NOT> ( <RELATIONAL EXP> )
119           | <COMPARISON>

120 <CHAR PRIM> ::= <CHAR VAR>
121           | <CHAR CONST>
122           | <MODIFIED CHAR FUNC>
123           | <CHAR INLINE DEF> <BLOCK BODY> <CLOSING> ;
124           | <CHAR FUNC HEAD> ( <CALL LIST> )
125           | ( <CHAR EXP> )

126 <CHAR FUNC HEAD> ::= <CHAR FUNC>
127                   | CHARACTER <SUB OR QUALIFIER>

128 <SUB OR QUALIFIER> ::= <SUBSCRIPT>
129                   | <BIT QUALIFIER>

130 <CHAR EXP> ::= <CHAR PRIM>
131           | <CHAR EXP> <CAT> <CHAR PRIM>
132           | <CHAR EXP> <CAT> <ARITH EXP>
133           | <ARITH EXP> <CAT> <ARITH EXP>
134           | <ARITH EXP> <CAT> <CHAR PRIM>

135 <ASSIGNMENT> ::= <VARIABLE> <=1> <EXPRESSION>
136           | <VARIABLE> , <ASSIGNMENT>

137 <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT>

```

ORIGINAL PAGE IS POOR

G-3

```

138             | <TRUE PART> <STATEMENT>
139 <TRUE PART> ::= <IF CLAUSE> <BASIC STATEMENT> ELSE
140 <IF CLAUSE> ::= <IF> <RELATIONAL EXP> THEN
141                | <IF> <BIT EXP> THEN
142 <IF> ::= IF
143 <DO GROUP HEAD> ::= DO ;
144                | DO <FOR LIST> ;
145                | DO <FOR LIST> <WHILE CLAUSE> ;
146                | DO <WHILE CLAUSE> ;
147                | DO CASE <ARITH EXP> ;
148                | <CASE ELSE> <STATEMENT>
149                | <DO GROUP HEAD> <ANY STATEMENT>
150                | <DO GROUP HEAD> <TEMPORARY STMT>
151 <CASE ELSE> ::= DO CASE <ARITH EXP> ; ELSE
152 <WHILE KEY> ::= WHILE
153                | UNTIL
154 <WHILE CLAUSE> ::= <WHILE KEY> <BIT EXP>
155                | <WHILE KEY> <RELATIONAL EXP>
156 <FOR LIST> ::= <FOR KEY> <ARITH EXP> <ITERATION CONTROL>
157                | <FOR KEY> <ITERATION BODY>
158 <ITERATION BODY> ::= <ARITH EXP>
159                | <ITERATION BODY> , <ARITH EXP>
160 <ITERATION CONTROL> ::= TO <ARITH EXP>
161                | TO <ARITH EXP> BY <ARITH EXP>
162 <FOR KEY> ::= FOR <ARITH VAR> =
163                | FOR TEMPORARY <IDENTIFIER> =
164 <ENDING> ::= END
165                | END <LABEL>
166                | <LABEL DEFINITION> <ENDING>
167 <ON PHRASE> ::= ON ERROR <SUBSCRIPT>
168 <ON CLAUSE> ::= ON ERROR <SUBSCRIPT> SYSTEM
169                | ON ERROR <SUBSCRIPT> IGNORE
170 <SIGNAL CLAUSE> ::= SET <EVENT VAR>
171                | RESET <EVENT VAR>
172                | SIGNAL <EVENT VAR>
173 <FILE EXP> ::= <FILE HEAD> , <ARITH EXP> )
174 <FILE HEAD> ::= FILE ( <NUMBER>
175 <CALL KEY> ::= CALL <LABEL VAR>
176 <CALL LIST> ::= <LIST EXP>
177                | <CALL LIST> , <LIST EXP>

```

G-4

```

174 <CALL ASSIGN LIST> ::= <VARIABLE>
175 | <CALL ASSIGN LIST> , <VARIABLE>

180 <EXPRESSION> ::= <ARITH EXP>
181 | <BIT EXP>
182 | <CHAR EXP>
183 | <STRUCTURE EXP>
184 | <NAME EXP>

185 <STRUCTURE EXP> ::= <STRUCTURE VAR>
186 | <MODIFIED STRUCT FUNC>
187 | <STRUC INLINE DEF> <BLOCK BODY> <CLOSING> ;
188 | <STRUCT FUNC HEAD> ( <CALL LIST> )

189 <STRUCT FUNC HEAD> ::= <STRUCT FUNC>

190 <LIST EXP> ::= <EXPRESSION>
191 | <ARITH EXP> # <EXPRESSION>

192 <VARIABLE> ::= <ARITH VAR>
193 | <STRUCTURE VAR>
194 | <BIT VAR>
195 | <EVENT VAR>
196 | <SUBBIT HEAD> <VARIABLE> )
197 | <CHAR VAR>
198 | <NAME KEY> ( <NAME VAR> )

199 <NAME VAR> ::= <VARIABLE>
200 | <LABEL VAR>
201 | <MODIFIED ARITH FUNC>
202 | <MODIFIED BIT FUNC>
203 | <MODIFIED CHAR FUNC>
204 | <MODIFIED STRUCT FUNC>

205 <NAME EXP> ::= <NAME KEY> ( <NAME VAR> )
206 | NULL
207 | <NAME KEY> ( NULL )

208 <NAME KEY> ::= NAME

209 <LABEL VAR> ::= <PREFIX> <LABEL> <SUBSCRIPT>

210 <MODIFIED ARITH FUNC> ::= <PREFIX> <NO ARG ARITH FUNC> <SUBSCRIPT>
211 <MODIFIED BIT FUNC> ::= <PREFIX> <NO ARG BIT FUNC> <SUBSCRIPT>
212 <MODIFIED CHAR FUNC> ::= <PREFIX> <NO ARG CHAR FUNC> <SUBSCRIPT>
213 <MODIFIED STRUCT FUNC> ::= <PREFIX> <NO ARG STRUCT FUNC> <SUBSCRIPT>
214 <STRUCTURE VAR> ::= <QUAL STRUCT> <SUBSCRIPT>
215 <ARITH VAR> ::= <PREFIX> <ARITH ID> <SUBSCRIPT>
216 <CHAR VAR> ::= <PREFIX> <CHAR ID> <SUBSCRIPT>
217 <BIT VAR> ::= <PREFIX> <BIT ID> <SUBSCRIPT>
218 <EVENT VAR> ::= <PREFIX> <EVENT ID> <SUBSCRIPT>

```

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

G-5

```

219 <QUAL STRUCT> ::= <STRUCTURE ID>
220 | <QUAL STRUCT> . <STRUCTURE ID>

221 <PREFIX> ::=
222 | <QUAL STRUCT> .

223 <SUBBIT HEAD> ::= <SUBBIT KEY> <SUBSCRIPT> (
224 <SUBBIT KEY> ::= SUBBIT

225 <SUBSCRIPT> ::= <SUB HEAD> )
226 | <QUALIFIER>
227 | <$> <NUMBER>
228 | <$> <ARITH VAR>
229 |

230 <SUB START> ::= <$> (
231 | <$> ( @ <PREC SPEC> ,
232 | <SUB HEAD> ;
233 | <SUB HEAD> ;
234 | <SUB HEAD> ,

235 <SUB HEAD> ::= <SUB START>
236 | <SUB START> <SUB>

237 <SUB> ::= <SUB EXP>
238 | *
239 | <SUB RUN HEAD> <SUB EXP>
240 | <ARITH EXP> AT <SUB EXP>

241 <SUB RUN HEAD> ::= <SUB EXP> TO

242 <SUB EXP> ::= <ARITH EXP>
243 | <# EXPRESSION>

244 <# EXPRESSION> ::= #
245 | <# EXPRESSION> + <TERM>
246 | <# EXPRESSION> - <TERM>

247 <=1> ::= =

248 <$> ::= $

249 <AND> ::= &
250 | AND

251 <OR> ::= |
252 | OR

253 <NOT> ::= ~
254 | NOT

255 <CAT> ::= ||
256 | CAT

257 <QUALIFIER> ::= <$> ( @ <PREC SPEC> )

258 <BIT QUALIFIER> ::= <$> ( @ <RADIX> )

```

G-6

```

259 <RADIX> ::= HEX
260           | OCT
261           | BIN
262           | DEC

263 <BIT CONST HEAD> ::= <RADIX>
264           | <RADIX> ( <NUMBER> )

265 <BIT CONST> ::= <BIT CONST HEAD> <CHAR STRING>
266           | TRUE
267           | FALSE
268           | ON
269           | OFF

270 <CHAR CONST> ::= <CHAR STRING>
271           | CHAR ( <NUMBER> ) <CHAR STRING>

272 <IO CONTROL> ::= SKIP ( <ARITH EXP> )
273           | TAB ( <ARITH EXP> )
274           | COLUMN ( <ARITH EXP> )
275           | LINE ( <ARITH EXP> )
276           | PAGE ( <ARITH EXP> )

277 <READ PHRASE> ::= <READ KEY> <READ ARG>
278           | <READ PHRASE> , <READ ARG>

279 <WRITE PHRASE> ::= <WRITE KEY> <WRITE ARG>
280           | <WRITE PHRASE> , <WRITE ARG>

281 <READ ARG> ::= <VARIABLE>
282           | <IO CONTROL>

283 <WRITE ARG> ::= <EXPRESSION>
284           | <IO CONTROL>

285 <READ KEY> ::= READ ( <NUMBER> )
286           | READALL ( <NUMBER> )

287 <WRITE KEY> ::= WRITE ( <NUMBER> )

288 <BLOCK DEFINITION> ::= <BLOCK STMT> <BLOCK BODY> <CLOSING> ;

289 <BLOCK BODY> ::=
290           | <DECLARE GROUP>
291           | <BLOCK BODY> <ANY STATEMENT>

292 <ARITH INLINE DEF> ::= FUNCTION <ARITH SPEC> ;
293           | FUNCTION ;

294 <BIT INLINE DEF> ::= FUNCTION <BIT SPEC> ;

295 <CHAR INLINE DEF> ::= FUNCTION <CHAR SPEC> ;

296 <STRUC INLINE DEF> ::= FUNCTION <STRUCT SPEC> ;

297 <BLOCK STMT> ::= <BLOCK STMT TOP> ;
298           | <BLOCK STMT TOP> ACCESS ;

299 <BLOCK STMT TOP> ::= <BLOCK STMT HEAD>

```

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

G-7

```

300          | <BLOCK STMT HEAD> EXCLUSIVE
301          | <BLOCK STMT HEAD> REENTRANT

302 <LABEL DEFINITION> ::= <LABEL> :

303 <LABEL EXTERNAL> ::= <LABEL DEFINITION>
304          | <LABEL DEFINITION> EXTERNAL

305 <BLOCK STMT HEAD> ::= <LABEL EXTERNAL> PROGRAM
306          | <LABEL EXTERNAL> COMPOOL
307          | <LABEL DEFINITION> TASK
308          | <LABEL DEFINITION> UPDATE
309          | UPDATE
310          | <FUNCTION NAME>
311          | <FUNCTION NAME> <FUNC STMT BODY>
312          | <PROCEDURE NAME>
313          | <PROCEDURE NAME> <PROC STMT BODY>

314 <FUNCTION NAME> ::= <LABEL EXTERNAL> FUNCTION

315 <PROCEDURE NAME> ::= <LABEL EXTERNAL> PROCEDURE

316 <FUNC STMT BODY> ::= <PARAMETER LIST>
317          | <TYPE SPEC>
318          | <PARAMETER LIST> <TYPE SPEC>

319 <PROC STMT BODY> ::= <PARAMETER LIST>
320          | <ASSIGN LIST>
321          | <PARAMETER LIST> <ASSIGN LIST>

322 <PARAMETER LIST> ::= <PARAMETER HEAD> <IDENTIFIER> )

323 <PARAMETER HEAD> ::= (
324          | <PARAMETER HEAD> <IDENTIFIER> ,

325 <ASSIGN LIST> ::= <ASSIGN> <PARAMETER LIST>

326 <ASSIGN> ::= ASSIGN

327 <DECLARE ELEMENT> ::= <DECLARE STATEMENT>
328          | <REPLACE STMT> ;
329          | <STRUCTURE STMT>

330 <REPLACE STMT> ::= REPLACE <REPLACE HEAD> BY <TEXT>

331 <REPLACE HEAD> ::= <IDENTIFIER>
332          | <IDENTIFIER> ( <ARG LIST> )

333 <ARG LIST> ::= <IDENTIFIER>
334          | <ARG LIST> , <IDENTIFIER>

335 <TEMPORARY STMT> ::= TEMPORARY <DECLARE BODY> ;

336 <DECLARE STATEMENT> ::= DECLARE <DECLARE BODY> ;

337 <DECLARE BODY> ::= <DECLARATION LIST>
338          | <ATTRIBUTES> , <DECLARATION LIST>

339 <DECLARATION LIST> ::= <DECLARATION>

```

G-8


```

340             | <DCL LIST ,> <DECLARATION>
341 <DCL LIST ,> ::= <DECLARATION LIST> ,
342 <DECLARE GROUP> ::= <DECLARE ELEMENT>
343             | <DECLARE GROUP> <DECLARE ELEMENT>
344 <STRUCTURE STMT> ::= STRUCTURE <STRUCT STMT HEAD> <STRUCT STMT TAIL>
345 <STRUCT STMT HEAD> ::= <IDENTIFIER> : <LEVEL>
346             | <IDENTIFIER> <MINOR ATTR LIST> : <LEVEL>
347             | <STRUCT STMT HEAD> <DECLARATION> , <LEVEL>
348 <STRUCT STMT TAIL> ::= <DECLARATION> ;
349 <STRUCT SPEC> ::= <STRUCT TEMPLATE> <STRUCT SPEC BODY>
350 <STRUCT SPEC BODY> ::= - STRUCTURE
351             | <STRUCT SPEC HEAD> <LITERAL EXP OR *> )
352 <STRUCT SPEC HEAD> ::= - STRUCTURE (
353 <DECLARATION> ::= <NAME ID>
354             | <NAME ID> <ATTRIBUTES>
355 <NAME ID> ::= <IDENTIFIER>
356             | <IDENTIFIER> NAME
357 <ATTRIBUTES> ::= <ARRAY SPEC> <TYPE & MINOR ATTR>
358             | <ARRAY SPEC>
359             | <TYPE & MINOR ATTR>
360 <ARRAY SPEC> ::= <ARRAY HEAD> <LITERAL EXP OR *> )
361             | FUNCTION
362             | PROCEDURE
363             | PROGRAM
364             | TASK
365 <ARRAY HEAD> ::= ARRAY (
366             | <ARRAY HEAD> <LITERAL EXP OR *> ,
367 <TYPE & MINOR ATTR> ::= <TYPE SPEC>
368             | <TYPE SPEC> <MINOR ATTR LIST>
369             | <MINOR ATTR LIST>
370 <TYPE SPEC> ::= <STRUCT SPEC>
371             | <BIT SPEC>
372             | <CHAR SPEC>
373             | <ARITH SPEC>
374             | EVENT
375 <BIT SPEC> ::= BOOLEAN
376             | BIT ( <LITERAL EXP OR *> )
377 <CHAR SPEC> ::= CHARACTER ( <LITERAL EXP OR *> )
378 <ARITH SPEC> ::= <PREC SPEC>
379             | <SQ DQ NAME>
380             | <SQ DQ NAME> <PREC SPEC>

```

G-9

```

381 <SQ DQ NAME> ::= <DOUBLY QUAL NAME HEAD> <LITERAL EXP OR *> )
382         | INTEGER
383         | SCALAR
384         | VECTOR
385         | MATRIX

386 <DOUBLY QUAL NAME HEAD> ::= VECTOR (
387         | MATRIX ( <LITERAL EXP OR *> ,

388 <LITERAL EXP OR *> ::= <ARITH EXP>
389         | *

390 <PREC SPEC> ::= SINGLE
391         | DOUBLE

392 <MINOR ATTR LIST> ::= <MINOR ATTRIBUTE>
393         | <MINOR ATTR LIST> <MINOR ATTRIBUTE>

394 <MINOR ATTRIBUTE> ::= STATIC
395         | AUTOMATIC
396         | DENSE
397         | ALIGNED
398         | ACCESS
399         | LOCK ( <LITERAL EXP OR *> )
400         | REMOTE
401         | <INIT/CONST HEAD> <REPEATED CONSTANT> )
402         | <INIT/CONST HEAD> * )
403         | LATCHED
404         | NONHAL ( <LEVEL> )

405 <INIT/CONST HEAD> ::= INITIAL (
406         | CONSTANT (
407         | <INIT/CONST HEAD> <REPEATED CONSTANT> ,

408 <REPEATED CONSTANT> ::= <EXPRESSION>
409         | <REPEAT HEAD> <VARIABLE>
410         | <REPEAT HEAD> <CONSTANT>
411         | <NESTED REPEAT HEAD> <REPEATED CONSTANT> )
412         | <REPEAT HEAD>

413 <REPEAT HEAD> ::= <ARITH EXP> #

414 <NESTED REPEAT HEAD> ::= <REPEAT HEAD> (
415         | <NESTED REPEAT HEAD> <REPEATED CONSTANT> ,

416 <CONSTANT> ::= <NUMBER>
417         | <COMPOUND NUMBER>
418         | <BIT CONST>
419         | <CHAR CONST>

420 <NUMBER> ::= <SIMPLE NUMBER>
421         | <LEVEL>

422 <CLOSING> ::= CLOSE
423         | CLOSE <LABEL>
424         | <LABEL DEFINITION> <CLOSING>

425 <TERMINATOR> ::= TERMINATE
426         | CANCEL

```

G-10

```

427 <TERMINATE LIST> ::= <LABEL VAR>
428 | <TERMINATE LIST> , <LABEL VAR>

429 <WAIT KEY> ::= WAIT

430 <SCHEDULE HEAD> ::= SCHEDULE <LABEL VAR>
431 | <SCHEDULE HEAD> AT <ARITH EXP>
432 | <SCHEDULE HEAD> IN <ARITH EXP>
433 | <SCHEDULE HEAD> ON <BIT EXP>

434 <SCHEDULE PHRASE> ::= <SCHEDULE HEAD>
435 | <SCHEDULE HEAD> PRIORITY ( <ARITH EXP> )
436 | <SCHEDULE PHRASE> DEPENDENT

437 <SCHEDULE CONTROL> ::= <STOPPING>
438 | <TIMING>
439 | <TIMING> <STOPPING>

440 <TIMING> ::= <REPEAT> EVERY <ARITH EXP>
441 | <REPEAT> AFTER <ARITH EXP>
442 | <REPEAT>

443 <REPEAT> ::= , REPEAT

444 <STOPPING> ::= <WHILE KEY> <ARITH EXP>
445 | <WHILE KEY> <BIT EXP>

```

H. SUMMARY OF OPERATORS

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

The information found in this appendix has been abstracted from chapter 6 of this specification.

H.1 ARITHMETIC OPERATORS*

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

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

H.2 CHARACTER OPERATOR*

OPERATOR	NAME	FORM
	concatenation	$\boxed{re} \ \ \boxed{sult} \ \longrightarrow \ \boxed{result}$

*Note that this table contains information found in Section 6.1.3.

H.3 BIT OPERATORS*

OPERATORS	NAME	BIT OPERATOR PRECEDENCE	FORM	COMMENTS
 CAT }	concatenation	1	B B	$\boxed{11101} \ \ \boxed{010} \ \longrightarrow \ \boxed{11101010}$
& AND }	logical product	2	B&B	Parallel operation bit by bit
 OR }	logical sum	3	B B	Parallel operation bit by bit
- NOT }	logical complement	Highest implied by syntax	$\neg B$	Parallel operation bit by bit;
<p>The following abbreviations apply: B = bit string or boolean</p>				

*Note that this table contains information found in Section 6.1.2

H.4 CONDITIONAL AND EVENT OPERATORS*

OPERATOR	NAME	CONDITIONAL PRECEDENCE	FORM	COMMENTS
& AND	logical product	1	C&C C AND C	True if both "C"s true
 OR	logical sum	2	C C C OR C	True if either "C" is true
~ NOT	logical complement	Highest implied by syntax	~C	Operand

The following abbreviations apply:

"C" = any conditional operand.

*Note that this table contains information found in Sections 6.2 and 6.3.

H.5 COMPARISON OPERATORS*

OPERATOR	USE	COMMENTS
> >= < <= ~> NOT> } ~< NOT< }	A > B A >= B A < B A <= B A ~> B A ~< B	} magnitude comparisons: apply only to unarrayed scalar and integer data A and B.
= NOT= ~= }	A=B A ~= B	} equality/inequality for general data A and B.

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

BIBLIOGRAPHY

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

Distribution

NASA

Johnson Space Center
Houston, Texas 77058

John L. Ford, BC28 (1)
Charles M. Grant, BM2 (2)
Gilbert C. Symons, BT3 (1)
John R. Garman, FR (3)

Kennedy Space Center
Florida 32815

J.R. Medlock, LV-CAP (1)

Aerospace Corporation

2350 E. El Segundo Blvd.
El Segundo, Calif. 90245

R.K. Luke (1)

CSDL

75 Cambridge Parkway
Cambridge, Mass. 02142

M. Hamilton, 73 (1)

IBM Electronics Systems Center

Bodle Hill Road
Owego, New York 13927

R.T. Smith, 002BC62 (1)

International Business Machines Corp.

1322 Space Park Drive
Houston, Texas 77058

J. Hoskins, MC69 (3)

Rockwell International

Space Division
12214 Lakewood Blvd.
Downey, Calif. 90241

E. Freddolino, FB-37 (1)
J. Ling, FA19 (1)

USAF

Unit Postal Office
Los Angeles, Calif. 90045

Capt. D.G. Keach, XRZT (1)

I. % MACROS

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

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

INDEX

ACCESS	3-14 to 3-18.1 4-13, 4-15, 4-16
to NAME variables	11-17, 11-27
active process	8-2
ALIGNED	4-9, 4-10, 4-13, 4-15, 4-16, 4-17, 11-20, 11-27
AND	6-7, 6-8 6-13, 6-21
apostrophe	4-7
argument type summary (chart)	6-37
arithmetic comparison	6-15
syntax diagram #32	
legal arithmetic comparisons	6-16
arithmetic conversion function	6-27, 6-37
syntax diagram #39	
<arith conversion>	6-6, 6-28
arithmetic expressions	6-3
syntax diagram #24	
<arith exp>	6-3, 6-15, 7-21, 8-12
syntax diagram #24	6-3
in subscript	5-12, 5-18
in type spec	4-19, 4-20
<arith exp>#	4-24, 6-28, 6-4
<arith inline>	11-3, 11-7
arithmetic literals	2-8
<arith %-macro>	11-5, 11-7
<arith operand>	6-3, 6-6
syntax diagram #25	
arithmetic operand	11-7
arith %-macro	
syntax diagram #25s	

<arith var>	5-16
ARRAY	4-13, 4-14
array dimension	5-17
array properties of expressions	6-12
array specification	4-14, 4-16, 4-17, 4-25, 4-26, 5-17
<array sub>	5-7, 5-8, 5-14
array subscripts syntax diagram #22	5-11
Array Subscripting	4-2, 5-7, 5-14, 7-10
arrayed <comparison>	6-18, 6-17
arrayed infix operations	6-12
arrayed operand comparison	6-20
Arrayness	5-17, 5-21
Assignment statements	7-5
event variables	6-22, 6-24
of NAME identifiers	11-18
of subscript expressions	5-18
ASSIGN	3-15, 7-9, 7-10
assign parameter	7-10
assignment	7-1
assignment statement syntax diagram #46	7-5
asterisk, use of	4-16, 4-21, 4-22, 4-24, 4-27, 5-11, 5-12
"*"	11-18
**	2-11

AT	5-11
AT <arith exp>	8-5
AT-partition	5-12, 5-13, 5-14, 5-15
<attributes>	4-12
factored <attributes>	4-12
AUTOMATIC	3-16, 3-16.1, 3-18, 4-13
basic statement	7-2
syntax diagram #44	
<basic statement>	7-24
BIT	4-21, 4-26, 6-31, 11-3
bit argument length	6-24
bit assignments	7-7
bit comparison	6-17
syntax diagram #33	
bit conversion function	6-31
syntax diagram #40	
<bit conversion>	6-8
bit expression	6-7
syntax diagram #26	
<bit exp>	4-26, 6-7, 6-8, 6-17, 6-34, 7-3, 7-17, 7-18
bit expression length	7-13
<bit inline>	11-3, 11-8
bit literals	2-9
<bit literal>	6-8, 6-9

bit operand	6-8
syntax diagram #27	
bit inline	
bit%-macro	
syntax diagram #27s	11-8
bit operator precedence	6-8
<bit %-macro>	11-5, 11-8
<bit-pseudo var>	5-6, 6-35, 7-7
<bit var>	6-9, 6-8
BIN	2-9, 6-32
@BIN	6-32, 6-34
blanks	2-13, 4-7, 4-24
Block delimiting statements	3-13
block name uniqueness	3-20
Block Templates	3-13
syntax diagram #6	3-11
BNF Grammar of HAL/S	Appendix G
BOOLEAN	4-19, 4-21, 4-26, 11-3
built-in functions	6-23, 11-1
built-in function names	2-6
built-in function parameters	6-24
BY	7-22

CALL Statement	7-9, 7-10
syntax diagram #47	7-9
with NAME	
syntax diagram #47s	11-32
call-by-reference	6-24, 7-10
call-by-value	6-24, 7-10
CANCEL statement	8-10
syntax diagram #58	8-9
cancellation	8-6, 8-7, 8-8, 8-9
CAT	6-7, 6-8, 6-10
catenation	6-10
channels	10-1
HAL/S character set	2-4
CHARACTER	4-21, 4-26, 6-33, 6-24, 7-10, 11-9
character argument length	6-24
character comparison	6-18
syntax diagram #34	
character conversion function	6-33
syntax diagram #41	
<char conversion>	6-11
character expression	6-10
syntax diagram #28	
character expression length	7-13
<char exp>	4-26, 6-10, 6-18, 6-32
character initialization	4-26
<char inline>	11-4, 11-9
character length	4-21
character literal	2-10, 4-7
<character literal>	2-10, 6-11
<character %-macro>	11-5, 11-9

character operand	6-11
syntax diagram #29	
char inline	
char %-macro	
syntax diagram #29s	11-9
character operator precedence	6-14
character string	6-10
character type	4-21
<char var>	6-11, 7-7
CLOSE Statement	3-19
syntax diagram #10	
CLOSE	7-12, 7-13
closing	3-4
<closing>	3-13, 3-19
code blocks	3-13
colon, use of	4-10, 5-14, 9-4
COLUMN	10-3, 10-7, 10-8, 10-9
comma, use of	4-7, 4-10, 5-11, 10-5
comments (imbedded)	2-13
<comparison>	6-13, 6-20
<compilation>	3-2, 3-20, 7-9
Component Subscripting	4-2, 5-7, 6-35, 7-7, 7-10, 10-11
component subscripts	5-11
syntax diagram #22	
<component sub>	5-7, 5-8, 5-16

COMPOOL	3-2, 3-14
COMPOOL block	3-13, 3-19
syntax diagram #5	3-10
<compool block>	4-16
COMPOOL block template	3-11
<compool header>	3-10
compool header statement	3-14
compool modules	3-1
<compool template>	4-16
<condition>	6-1, 6-14, 7-4, 7-17, 7-20, 7-22
conditional expression	6-1
syntax diagram #30	6-13
conditional operand	6-14
syntax diagram #31	
<conditional operand>	6-13
CONSTANT	4-23
conversion	6-24
conversion functions	
summary of argument types	6-37
cyclic execution	8-4, 8-6
Data declarative attributes	4-13
syntax diagram #15	
data declarative <attributes>	4-12
Data Manipulation	6-1

Data NAME identifiers	11-18
Data referencing	5-1
Data Sharing and the UPDATE Block	8-19
data types	1-2
DEC	2-9, 6-32
@DEC	6-32, 6-34
DECLARE Statement	4-12
syntax diagram #14	
with NAME	
syntax diagram #14s	11-16
<declare statement>	4-3, 11-42
declare group	3-4, 3-6, 4-1
syntax diagram #11	4-3
with EQUATE	
syntax diagram #115	11-42
<declare group>	3-12, 5-2
Declarations of Temporaries	11-13
Declaration of NAME temporaries	11-24
DENSE/ALIGNED	4-15, 11-17
DENSE	4-9, 4-10, 4-13, 4-16, 4-17, 7-10, 7-10.1, 7-11, 11-17, 11-20, 11-22
DEPENDENT	8-6, 8-12
dependent processes	8-2
DO	11-12
DO statement	7-15
syntax diagram #50	
<do statement>	7-14, 7-23
DO CASE statement	7-16
syntax diagram #51	
DO...END statement group	7-14
syntax diagram #49	

DO...END	7-23, 7-24, 7-25, 11-15
DO...END statement TEMPORARY statement syntax diagram #49s	11-12
DO FOR	11-15
Discrete DO FOR Statement syntax diagram #53	7-19
discrete DO FOR with loop TEMPORARY variable index syntax diagram #53s	11-14
iterative DO FOR	7-21
DO WHILE and UNTIL statements syntax diagram #52	7-17
DO UNTIL	7-18
DO WHILE	11-30
DOUBLE	4-19, 4-20, 6-38, 7-6
double precision	6-15
double quotes	4-5
ELSE	7-3, 7-4, 7-16
dangling ELSE	7-4
END	7-23
END statement syntax diagram #55	7-23
<end statement>	7-14, 7-23

EQUATE Statement	11-40
syntax diagram #80	11-40
errors	9-1
system-defined	
user-defined	
error code	9-1, 9-5, 9-7
error environment	9-1
error groups	9-1
error number	9-7
Error precedence (chart)	9-6
<error spec>	9-3, 9-4, 9-5
Error Recovery	1-2, 7-1
Error Recovery Executive (ERE)	8-9, 9-1, 9-4, 9-7
EVENT	4-18, 4-19, 4-21, 4-26, 11-13, 11-23
event change point	8-5, 8-7, 8-8, 8-15
Event Control	8-15
event expression	6-1
syntax diagram #36	6-21
<event exp>	6-21, 8-13
event infix operator precedence	6-21, 6-22
event operand	6-22
syntax diagram #57	
<event operand>	6-21
<event var>	6-9
latched	8-16
unlatched	8-16

EVERY <arith exp>	8-6
EXCLUSIVE	3-15 to 3-18.1
executable statements	7-1
EXIT statement syntax diagram #56	7-24
EXIT	7-25
explicit conversion functions	6-26, 6-27, 6-31, 6-33
explicit type conversion	6-24, 6-38
exponent	7-6
<exponents>	2-8
exponentiation	2-12
<expression>	6-1, 7-5, 7-12
external procedure	3-1
EXTERNAL	3-12, 11-40
extended character set	2-4
FALSE	2-9, 4-26, 7-4, 7-17, 8-16
father	8-2
FILE	10-11
<file exp>	10-10, 10-11
FILE statement syntax diagram #68	10-10
paged file	10-2
unpaged file	10-2

flow control	7-1
flow of execution	3-5, 3-7
flow path	2-3
format	1-1
formal parameters	3-15, 3-17, 4-16, 4-18
FUNCTION	3-2, 3-5, 3-7, 3-8, 3-17, 3-20, 4-18, 7-12, 11-3, 11-19, 11-20
<function>	7-12
FUNCTION block syntax diagram #3	3-6
<function block>	3-17, 6-23, 6-24
FUNCTION block template	3-11
Function header statement syntax diagram #9	3-17
<function header>	3-17, 4-4
function modules	3-1
<function template>	3-12, 3-17, 6-23
user defined	6-24, 6-27
GO TO	7-24, 7-25,
GO TO Statement syntax diagram #56	7-24

Hardware discretetes	8-15
header statements syntax diagram #17	3-14
HEX	2-9, 6-32
@HEX	6-32, 6-34
identifiers	2-5, 2-7
<identifier>	3-15, 3-17, 4-3, 4-10, 4-11, 4-25, 11-40, 11-41
identifier generation with REPLACE	4-7.1
identifiers with NAME attribute	11-16
IF	7-2, 6-1
IF statement syntax diagram #45	7-3
IGNORE	9-4
implicit conversion	7-11, 7-13
implicit type conversion	6-26, 7-6
IN <arith exp>	8-5
independent processes	8-2
infix operators (chart)	6-3, 6-4 6-4
INITIAL	4-23
initial list	4-23
<initial list>	4-24, 4-25, 4-26, 4-27

initialization	4-13
<initialization>	4-14, 4-15, 4-16, 4-23, 11-17, 11-27
partial initialization	4-27
initialization specification syntax diagram #18	4-23
initiation	8-2
inline function	11-1
<inline function> syntax diagram #69	11-2
Inline function blocks	11-2
input argument	7-10, 7-11
input/output	7-1
I/O channel number	10-6
I/O control function syntax diagram #67	10-8
<I/O control>	10-3, 10-4, 10-6, 10-7
random access I/O	10-1, 10-10
I/O statements	10-1
input parameters	3-15, 3-17, 6-24
INTEGER	4-19, 4-20, 4-24, 6-27, 6-28
integer	7-6
integer-valued literal	2-8
Introduction	1-1
Iterative DO FOR statement syntax diagram #54	7-21
iterative DO FOR with loop TEMPORARY variable index syntax diagram #54s	11-14

keywords	2-6
<label>	2-7, 3-8, 3-10, 3-11, 3-19, 3-20, 4-18, 6-23, 7-2, 7-3, 7-9, 7-23, 7-24, 7-25, 8-5, 8-14
<%label>	11-5
Label declarative attributes syntax diagram #16 with NAME syntax diagram #16s	4-18 11-19
label declarative <attributes>	4-12
Label Name identifiers	11-20
LATCHED	4-13, 4-15, 4-21, 4-26
latched event	8-15
LINE	10-3, 10-7, 10-8, 10-9
linear array	4-14
literals	2-5, 2-8
literal zero	7-6
LOCK	3-16.1, 3-18.1, 4-13, 4-15, 4-16, 7-10, 8-19
LOCK(*)	8-19
lock group	7-10
Loop TEMPORARY variable syntax diagram #53s syntax diagram #54s	11-15 11-14 11-14

loop variable	7-19, 7-20, 7-22
machine units	8-3
%-macro	2-6, 11-1, 11-6
arith	11-5
bit	11-5
char	11-5
struct	11-5
typeless	11-5
%-macro references	11-4
syntax diagram #70	11-5
%-macro CALL statement	11-11
syntax diagram #71	
<%-macro call statement>	11-5, 11-11
major structure	4-13, 4-16, 5-3, 5-9, 5-20
mantissa	7-6
Matrix	5-21, 7-6
MATRIX	4-19, 4-20, 4-25, 5-15, 6-27, 6-29
maximum index value	5-12
minor structure	4-8, 4-10, 4-17, 5-3, 5-20
minor structure node	7-11, 10-11
MU	8-3
multiple copies	4-22, 5-9, 5-13, 5-17, 5-20, 7-11

name scope	3-20, 4-3, 4-4, 4-6, 4-10, 4-12, 7-9
Name Scope Rules	3-20
name uniqueness	4-10.1, 4-12
NAME facility	11-16
NAME argument passage	11-31
NAME assign syntax diagram #74	11-29
NAME assignment statement syntax diagram #75	11-29 11-30
<name assign>	11-30, 11-32, 11-34
NAME assignment	11-35
NAME attribute syntax diagram #14s	11-16
NAME attribute in structure templates syntax diagram #13s	11-22
NAME attribute	11-23, 11-24
NAME conditional expression syntax diagram #76	11-30
NAME data and structures	11-34
NAME facility	11-16
name identifier syntax diagram #16s	11-19
NAME identifier label	11-20
NAME identifier	11-20, 11-23, 11-24, 11-27
simple NAME identifiers	11-25
dereferenced use of simple NAME identifiers	11-25

<NAME id>	11-29
NAME initialization attribute syntax diagram #79	11-33
in I/O operations	11-39
NAME reference syntax diagram #73	11-26
<name reference>	11-30, 11-34
NAME (NULL)	11-28
Null NAME values	11-26
NAME pseudo-function	11-29
Referencing NAME values	11-26
NAME structure	11-34, 11-38
NAME variable	11-17
natural sequence	4-24, 5-20, 6-28
nested blocks	3-8
Nested Structure Template References	11-23
NONHAL	4-18
normal function syntax diagram #38	6-6, 6-24 6-23
<normal function> with NAME Syntax Diagram #77	6-24, 11-31
NOT	6-8, 6-9, 6-14, 6-19, 6-20
Null	4-7, 7-6, 11-28
Null character literal	2-10
Null field	10-5
Null statement syntax diagram #56	7-24

Null string	4-21
<number>	4-18, 6-6
object modules	3-1
OCT	2-9, 6-32
@OCT	6-32, 6-34
OFF ERROR	9-2, 9-4
ON ERROR	7-2, 7-3, 9-1, 9-4, 9-5
ON ERROR Statement syntax diagram #63	9-2 9-3
ON <event exp>	8-5
one-dimensional source format	2-11
operand	5-20, 6-1
OR	6-7, 6-8, 6-13, 6-21
packing attribute	4-10
PAGE	10-3, 10-7, 10-8, 10-9
parametric replace reference syntax diagram #12.1	4-7 4-6
parentheses	2-12, 4-7
partitioning SUBBIT subscripts	6-36
pointer	11-16

<power>	2-8
precedence	6-5
precision specifier syntax diagram #43	6-38
<precision>	6-6, 6-27, 6-29, 6-38
precision	6-38
precision conversion	7-6
primal process	8-2
HAL/S Primitives	2-1, 2-5
PRIORITY	8-6
PROCEDURE	3-2, 3-5, 3-7, 3-8, 3-15, 3-20, 7-9, 7-12, 11-3, 11-19
PROCEDURE block syntax diagram #3	3-4 3-6
PROCEDURE block template	3-11
Procedure Header Statement syntax diagram #8	3-15 3-15
<procedure header>	4-4
<procedure template>	3-12, 3-15, 7-10
process events	8-18
<process event>	11-20
<process-event name>	2-7, 6-8, 6-9, 6-22
process queue	8-4, 8-10

Program	3-2, 3-14, 6-22, 7-12, 8-2, 11-19, 11-20
PROGRAM block syntax diagram #2	3-4
<program block>	3-5
Program block template	3-11
program complex	3-1
program header	3-4
program header statement	3-14
qualified structure	4-22, 5-3, 5-4
<radix>	6-31, 6-32, 6-33, 6-34
random-access I/O	10-1, 10-10
READ	6-36, 10-2, 10-4, 10-8
READ and READALL statements syntax diagram #65	10-3
READALL	6-36, 10-2, 10-4, 10-8
ready	8-2
ready state	8-4
real time	3-8, 7-1, 11-3
real time control	1-2, 8-1
Real Time Executive	8-1
real time processes	8-2

REENTRANT	3-15, 3-16, 3-18
referencing simple variables	5-2
referencing structures	5-3
regular expression	6-1
syntax diagram #23	6-2
REMOTE	3-14.1, 4-13, 4-15, 7-10, 7-11, 11-23
REPEAT	7-25, 8-6
REPEAT statement	7-24
syntax diagram #56	
REPEAT EVERY	8-5
REPLACE	4-5, 4-6, 4-7,
;	11-2
syntax diagram #12	4-4
<replace statement>	4-3, 11-42
reraveling	5-20, 6-28
reserved words	2-5, 2-6
RESET	8-16
syntax diagram #62	8-15
restricted character set	2-4
RETURN	3-17, 3-19, 7-13
RETURN statement	7-12
syntax diagram #48	
RIGID	3-14, 3-14.1, 4-9, 4-10, 4-13, 4-15, 4-16, 4-17
rounding	6-28
row and column dimensions (Matrix)	4-20, 6-29
RTE	8-1, 8-2, 8-3, 8-4, 8-11, 8-15

RTE-clock	8-3, 8-5, 8-6, 8-12
run time errors	9-1
S	5-9
S;	5-9
SCALAR	4-19, 4-20, 4-24, 6-27, 6-28, 7-6
scalar valued literals	2-8
SCHEDULE	8-5, 8-7
SCHEDULE statement syntax diagram #57	8-4
semi colon, use of	5-13, 7-24, 10-4, 10-5
SEND ERROR syntax diagram #64	9-1 9-7
SET statement syntax diagram #62	8-16 8-15
SET, SIGNAL, and RESET syntax diagram #62	8-15
SET, SIGNAL, and RESET summary	8-17
Sequential I/O	10-1
Sequential I/O statements	10-2
shaping functions	6-26
SIGNAL statement syntax diagram #62	8-16 8-15
simple index	5-12
single precision	6-15

SINGLE	4-19, 4-20, 6-29, 7-6
SINGLE (default)	4-20
SKIP	10-3, 10-7, 10-8, 10-9
son	8-2
source macro	4-4, 4-6
source modules	3-1
source text	2-13
stall	8-2
stall state	8-4, 8-6
statement	3-4
<statement>	7-2, 9-5
STATIC	3-16, 3-16.1, 3-18, 4-13
STATIC (default)	4-14
STATIC/AUTOMATIC	3-16.1, 4-14, 11-17, 11-20
STRUCTURE	4-8, 4-10, 4-19, 4-21, 4-22
structure assignments	7-8
structure comparison syntax diagram #35	6-19
subscript construct syntax diagram #21	5-8
structure copies	4-27, 6-19
structure copy dimensions	5-17
structure copy specification NAME	11-18
structure expression Syntax Diagram #29.1	6-12
struct inline	11-10
struct % macro	11-10
syntax diagram #29.1s	11-10

<structure exp>	6-12, 6-19, 7-8
<struct inline>	11-4, 11-10
<struct %-macro>	11-5, 11-10
structure subscripts syntax diagram #22	5-11
structure template statement syntax diagram #13	4-9
structure template tree diagram	4-8 4-8
structure template statement with NAME syntax diagram #13s	11-22
<structure template>	4-3, 4-10, 4-16, 4-17, 11-42
structure terminal	4-10, 4-13, 4-16, 7-11
structure terminal refernces	11-34
subscripting structure terminals	11-36
structures containing NAME terminals	11-38
unarrayed structure terminal array structure terminal	5-9
structure type	11-18
structure types	4-27
Structure Subscripting	4-2, 5-7, 5-13
<structure sub>	5-7, 5-8, 5-13
non qualified structure variable declaration	4-11
<structure var name>	5-3
<structure var>	6-12

SUBBIT	5-6, 6-35
SUBBIT pseudo-variable syntax diagram #42	6-35
Subscripting syntax diagram #19	5-5
subscripting classes	5-7
component subscripting	5-8
legal subscript combinations	5-8
<subscript>	6-6, 6-27, 6-28, 6-29, 6-31, 6-34
<sub exp>	5-13, 5-14
<sub id>	11-26, 11-27, 11-33
<sub name id>	11-26, 11-27, 11-33
subscript line	2-11
syntax diagrams	2-1, 2-2
syntax diagram summaries	2-1, 2-2, Appendix A
SYSTEM	9-4
systems language features	11-1
system-defined erros	9-7
"T"	6-4
TAB	10-3, 10-7, 10-8
TASK	3-5, 3-14, 4-18, 6-22, 7-12, 8-2, 8-6, 11-19, 11-21
TASK block syntax diagram #3	3-6
task block	3-4
<task block>	4-18
task header statement	3-14

<template name>	2-7, 4-8, 5-3
TEMPORARY	11-12, 11-13
TEMPORARY statement	
syntax diagram #49s	11-12
syntax diagram #72	11-13
Temporary Variables	11-12
regular TEMPORARY variables	11-12
TERMINATE statement	8-11
syntax diagram #59	
<text>	4-4, 4-5, 4-6
THEN	7-3, 7-4
timing considerations	8-3
timing lines	8-15
TO	5-11, 7-22
TO-partition	5-12, 5-13, 5-14, 5-15
transpose	6-4
tree organization	6-19, 7-8
TRUE	2-9, 4-26, 8-16
two dimensional Source Format	2-11
type conversion	4-25
type specification	4-19
syntax diagram #17	
<type spec>	4-13, 4-14, 4-16, 4-17, 4-19, 4-20, 4-22, 11-3
typeless % macro	11-11

unary minus	6-5
unary plus	6-5
unit of compilation	3-2
syntax diagram #1	3-2
unlatched event	8-15
unqualified structure	4-22, 5-3, 5-4
unraveling	5-20, 6-28, 6-30
UNTIL	6-1, 7-18, 7-19, 7-20, 8-6, 8-7
UNTIL <arith exp>	8-7, 8-8
UPDATE	3-5, 3-7, 3-14, 8-19
UPDATE block	3-8
syntax diagram #4	
update block	3-4
<update block>	3-9
<update header>	3-8
update header statement	3-14
UPDATE PRIORITY statement	8-14
syntax diagram #61	
variable	5-5
syntax diagram #20	
unarrayed simple variable	5-9
arrayed simple variable	
<variable>	7-5, 7-6, 10-4, 11-40
<\$var name>	2-7, 5-2, 5-3

<\$var name> (interpretation table	5-9
<\$var>	5-17
simple variable	4-13, 4-16, 5-2
VECTOR	4-19, 4-20, 4-25, 5-15, 5-21, 6-27, 6-29, 7-6
vector length	4-20
wait	8-2
WAIT statement syntax diagram #60	8-12
WAIT	8-12
WAIT FOR	8-13
WAIT FOR DEPENDENT	8-13
WAIT UNTIL	8-12
WHILE	6-1, 7-19, 7-20, 7-22, 8-6
WHILE <event exp>	8-7
WRITE syntax diagram #66	6-2, 10-2, 10-8 10-6
/*...*/, use of	2-13
#, use of	5-11, 5-12
¢, use of	2-4, 2-10, 2-10.1
@, use of	2-4, 4-5, 4-7.1

Page intentionally left blank

UPDATE SHEET

Enclosed with each update package you receive is a new update sheet listing the new Version number, the affected pages, and the date of the update.

As you receive each update, replace the old update sheet with the new one. It is important that you refer to the most recent Version number on the update sheet whenever you correspond with Intermetrics concerning this document.

VERSION	AFFECTED MATERIAL	DATE
IR-61-4	This version of the Language Specification constitutes a final revision of the document. Many changes have been made and the user should review the entire document.	6/15/74

COMMENTS AND CHANGE OF ADDRESS

HAL/S users are encouraged to comment on this document and to note any discrepancies that they may encounter. The form below is provided for this purpose.

If the user believes that he is not receiving updates to the document, he should indicate the current Version number of the document, and check the update box below.

NAME _____ COMPANY _____

ADDRESS _____

BUSINESS PHONE _____ BUSINESS MAIL CODE _____

DOCUMENT VERSION NUMBER _____ UPDATES

COMMENTS _____

If the user is changing his address, but would like to continue receiving updates to this document, he should fill out his new address below, and the old address, along with the document Version number, above.

New Company _____

New Address _____

Business Mail Code _____ Business Phone _____