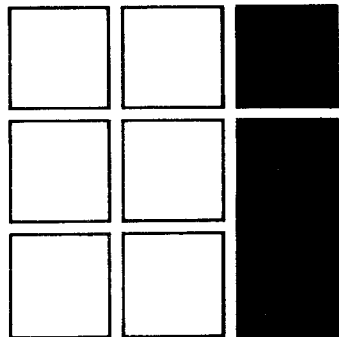


HAL/S LANGUAGE FORMS

5 April 1973



INTERMETRICS

HAL/S LANGUAGE FORMS

5 April 1973

Approved by: _____

F. H. Martin

Date: _____

FOREWORD

This document has been prepared by Intermetrics, Inc.
under Purchase Order #M3M8XMx-48300 for Rockwell International.

PREFACE

The purpose of this document is to present the acceptable forms of the HAL/S language in terms of a compendium of syntax diagrams. The diagrams and a condensation of syntax rules have been abstracted from the HAL/S Language Specification, while a larger set of examples has been included to illustrate the use of HAL/S.

The organization of this document follows the HAL/S Language Specification format exactly from Section 2 through Section 10 to allow easier reference. Section 1 provides an overview of the HAL/S language, Section 2 explains the manner in which the syntax diagrams may be read, and includes other format information such as the accepted character set, etc. Sections 3 through 10 present the HAL/S syntax as well as illustrative examples. A series of appendices are included which list keywords, built-in functions, and conversion functions, and summarizes several classes of HAL/S operations. Additionally, a more complex demonstration program is provided.

It is hoped that this document will serve as a reference for the HAL/S student, and an interim handbook until publication of the HAL/S Programmers Reference Manual.

TABLE OF CONTENTS

	<u>Page</u>
1. BRIEF DESCRIPTION OF HAL/S	1
1.1 Source Input/Source Listing	1
1.2 Data Types and Computations	2
1.3 Real-Time Control	3
1.4 Program Reliability	3
2. SYNTAX DIAGRAMS AND HAL/S PRIMITIVES	5
2.1 The HAL/S Syntax Diagram	5
2.2 The HAL/S Character Set	8
2.3 HAL/S Primitives	10
2.3.1 Reserved Words	10
2.3.2 Identifiers	10
2.3.3 Literals	11
2.4 Single Line and Multiple Line Source Text	14
2.5 Other Aspects	16
3. HAL/S BLOCK STRUCTURE AND ORGANIZATION	17
3.1 The Unit of Compilation	17
3.2 The PROGRAM Block	19
3.3 The PROCEDURE, FUNCTION, and TASK Blocks	21
3.4 The UPDATE Block	23
3.5 The COMPOOL Block	24
3.6 PROCEDURE, FUNCTION, and COMPOOL Templates	25
3.7 Block Delimiting Statements	27
3.7.1 Simple Header Statements	27
3.7.2 The Procedure Header Statement	28
3.7.3 The Function Header Statement	29
3.7.4 The CLOSE Statement	30

3.8	Name Scope Rules	31
4.	DATA AND LABEL DECLARATIONS	33
4.1	The Declare Group	33
4.2	The Replace Statement	35
4.3	The Structure Template	36
4.4	The DECLARE Statement	39
4.5	Label Declarative Inflections	40
4.6	Data Declarative Inflections	41
4.7	Type Specification	43
4.8	Initialization	45
5.	DATA REFERENCING CONSIDERATIONS	47
5.1	Referencing Simple Variables	47
5.2	Referencing Structures	47
5.2.1	Unqualified Structures	47
5.2.2	Qualified Structures	49
5.3	Subscripting	50
5.3.1	Kinds of Subscripting	51
5.3.2	Forms of Subscripting	54
5.3.3	The Arrayness of Variables and Expressions	56
5.4	The Natural Sequence of Elements	57
5.4.1	The Natural Sequence of Major and Minor Structures	57
5.4.2	The Natural Sequence of Simple Variables and Structure Terminals	57
6.	DATA MANIPULATION AND EXPRESSIONS	59
6.1	Regular Expressions	59
6.1.1	Arithmetic Expressions	60
6.1.2	Bit Expressions	62
6.1.3	Character Expressions	63

6.1.4	Regular Expression Operands	64
6.1.4.1	Arithmetic Operands	64
6.1.4.2	Bit Operands	65
6.1.4.3	Character Operands	66
6.1.5	Array Properties of Expressions	67
6.2	Conditional Expressions	68
6.2.1	Arithmetic Comparisons	69
6.2.2	Bit Comparisons	70
6.2.3	Character Comparisons	71
6.2.4	Structure Comparisons	72
6.2.5	Comparisons Between Arrayed Operands	72
6.3	Event Expressions	73
6.4	Normal Functions	74
6.5	Explicit Type Conversions	75
6.5.1	Arithmetic Conversion Functions	75
6.5.2	The Bit Conversion Function	77
6.5.3	The Character Conversion Function	78
6.5.4	The SUBBIT Pseudo-Variable	79
6.6	Explicit Precision Conversion	80
7.	EXECUTABLE STATEMENTS	81
7.1	Basic Statement Definition	81
7.2	The IF Statement	82
7.3	The Assignment Statement	83
7.4	The CALL Statement	84
7.5	The RETURN Statement	85
7.6	The DO...END Statement Group	86
7.6.1	The Simple DO Statement	86
7.6.2	The DO CASE Statement	87
7.6.3	The DO WHILE and DO UNTIL Statements	89
7.6.4	The Discrete DO FOR Statement	90

7.6.5	The Iterative DO FOR Statement	92
7.6.6	The END Statement	93
7.7	Other Basic Statements	94
8.	REAL TIME CONTROL	95
8.1	Real Time Processes and the RTE	95
8.2	Timing Considerations	95
8.3	The SCHEDULE Statement	96
8.3.1	The Simple SCHEDULE Statement	96
8.3.2	The Cyclic SCHEDULE Statement	98
8.4	The CANCEL Statement	99
8.5	The TERMINATE Statement	100
8.6	The WAIT Statement	100
8.7	The UPDATE PRIORITY Statement	102
8.8	Events and SIGNAL Statement	103
8.9	Process-Events	104
8.10	Data Sharing and the Update Block	104
9.	ERROR RECOVERY AND CONTROL	107
9.1	The ON ERROR Statement	107
9.2	The SEND ERROR Statement	108
10.	INPUT/OUTPUT STATEMENTS	111
10.1	Sequential I/O Statements	111
10.1.1	The READ and READALL Statements	111
10.1.2	The WRITE Statement	113
10.1.3	I/O Control Functions	114
10.2	Random Access I/O - The FILE Statement	115

APPENDICES:

A. HAL/S Keywords	117
B. HAL/S Built-In Functions	119
C. Summary of HAL/S Operations	121
D. Conversion Functions	129
E. Sample Program Listing	131

1.0 BRIEF DESCRIPTION OF HAL/S

HAL/S is a programming language developed by Intermetrics, Inc. for the Space Shuttle. It is intended to satisfy the requirements for both on-board and support software. The language contains features which provide for real-time control, vector-matrix and array data handling, and bit and character string manipulations.

1.1 Source Input/Source Listing

A singular feature of HAL is that it accepts and lists source code in a multi-line format, corresponding to the natural notation of ordinary algebra. An equation which involves exponents and subscripts will be written, for example, as

$$C = (X A^2 + Y B^{2.3/2})$$

I J K

instead of (as in FORTRAN or PL/1)

$$C(I) = (X*A(J)**2+Y*B(K)**2)**(3./2)$$

HAL also permits an optional single-line input format; its construction is similar to FORTRAN, with some minor changes; thus

$$C$I = (X A$J**2+Y B$K**2)**3/2$$

HAL/S source code may be input on cards or by data terminal. The input stream is free-form in that, for the most part, card or carriage column locations have no meaning; statements are separated simply by semi-colons.

In an effort to increase program reliability and promote HAL/S as a more direct communications medium between specifica-

tions and code, the HAL/S program listing is annotated with special marks. Vectors, matrices and arrays of data are instantly recognized by bars, stars and brackets. Thus, a vector becomes \bar{V} , a matrix \bar{M} , and an array [A]. Further, bit strings appear with a dot, i.e., \bar{B} and character strings with a comma, \bar{C} . With these special marks as aids, the source listing is more easily understood and serves as an important step toward self-documentation. In addition to data marks the HAL/S output listing has been standardized; logical paragraphs, or blocks of code, are automatically indented so that dependence of one block on another may be seen clearly.

HAL/S is a higher-order language, designed to allow programmers, analysts and engineers to communicate with the computer in a form which approximates 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.

1.2 Data Types and Computations

HAL/S provides facilities for manipulating a number of different data types. Arithmetic data may be declared as scalar, vector, matrix or integer (whole number). Individual bits may be treated as Boolean quantities or grouped together in strings. The language permits the user to manipulate character strings, via special instructions. Organizations of data may also be constructed; multi-dimensional arrays of any single type can be formulated, partitioned, and used in expressions. A hierarchical organization called a structure can be declared, in which related data of different types may be stored and retrieved as a unit or by individual reference.

The arithmetic data types together with the appropriate operators and built-in functions constitute a useful mathematical subset. HAL/S may be used in a straightforward manner as a "vector-matrix" language in implementing large portions of both on-board and support software. For example, a simplified equation of motion might appear as

$$\bar{A} = \bar{B} * \bar{ACC};$$

$$\bar{G} = -\bar{MU} \bar{UNIT}(\bar{R}) / \bar{R} \cdot \bar{R};$$

$$\bar{VDOT} = \bar{A} + \bar{G};$$

$$\bar{RDOT} = \bar{V};$$

where the matrix B^* transforms acceleration from measurement to reference coordinates.

By combining data types within expressions and utilizing both implicit and explicit conversions from one type to another, HAL/S may be applied to a wide variety of problems with a powerful and versatile capability.

1.3 Real-Time Control

HAL/S is a real-time control language; that is, certain defined blocks of code called programs and tasks can be scheduled based on time and/or the occurrence of anticipated events. These events may include external interrupts, specific data conditions, and programmer-defined software signals. Undesirable or unexpected events, such as abnormal conditions, may be handled by instructions which enable the programmer to specify appropriate action.

1.4 Program Reliability

Program reliability is enhanced when a software system can create effective isolation for various subsections of code as well as maintain and control commonly used data. HAL/S is a block-oriented language in that a block of code can be established with locally defined variables that cannot be altered by sections of program located outside the block. Independent blocks can be compiled and run together with communication among the programs permitted through a centrally managed and highly visible data pool. For a real-time environment, HAL/S couples these precautions with a protection mechanism which prevents, by programmer directive, the unauthorized or untimely use of commonly shared data and/or subroutines.

These measures cannot in themselves ensure total software reliability but HAL/S does offer the tools by which many anticipated problems, especially those prevalent in real-time control, can be isolated and solved.

2.0 SYNTAX DIAGRAMS AND HAL/S PRIMITIVES

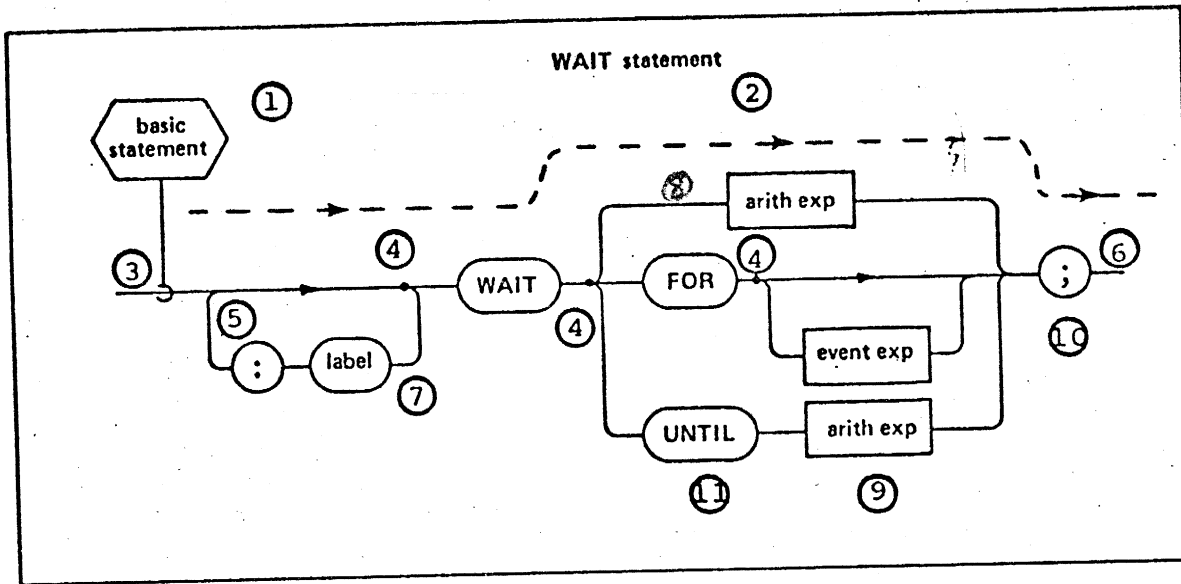
In this Specification, the syntax of the HAL/S language is represented in the form of syntax diagrams. These are to be read in conjunction with the associated sets of semantic rules. Together the two provide a complete, unambiguous description of the language. The syntax diagrams are mutually dependent in that syntactical elements referenced in some diagrams are defined in others. There are, however, a basic set of elements for which no definition is given. These are the so-called "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 the format of HAL source text which impact upon the meaning of the diagrams are also discussed briefly.

2.1 The HAL/S Syntax Diagram

Syntax diagrams are a flow-diagram like means of representing the formal grammar of a language. By tracing the paths on the diagrams, various examples of the language construct represented may be generated. In the context of HAL/S it is this generational aspect of the syntax diagrams which is emphasized. It is stressed that although the flow diagrams presented in this Language Forms manual are logically complete, they are not meant to be viewed as constituting a "working" grammar (that is, as an analytical tool for compiler construction). Rather they are to be viewed as purely instructional in nature.

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



RULES:

1. In every diagram there is a syntactical element being defined. The name of the element being defined appears in the hexagonal box (1). The title of the syntax diagram (2) is usually a discursive description of the syntactical element. In the case illustrated, the language construct depicted is a particularization of the syntactical element defined (a "WAIT statement" is an example of (1)).
2. To generate samples of the construct, the line 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 line (6) is reached.
3. The line is moved along until a black dot (4) is arrived at. 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 element it represents is added to the right of the sequence of

elements generated by moving along the line. For example, moving along the path denoted by the dotted line ⑧ generates the sequence "WAIT <arith exp>;" (see Rule 7.).

6. Boxes with squared corners such as ⑨ represent syntactical elements defined in other diagrams. Circular boxes such as ⑩, or boxes with circular ends, such as ⑪, represent HAL/S primitives.
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.

2.2 The HAL/S Character Set

The HAL/S character set consists of the 26 alphabetic characters, the numerals zero through nine, and certain special characters. The restricted character set is the set necessary for the construction of the HAL/S primitives to be described. The extended character set adds to the restricted set certain extra special characters legal in places like comments and character literals, and used chiefly for the purpose of compiler listing annotation.

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

alphabetic and numeric	special character
<p>A a</p> <p>B b</p> <p>C c</p> <p>D d</p> <p>E e</p> <p>F f</p> <p>G g</p> <p>H h</p> <p>I i</p> <p>J j</p> <p>K k</p> <p>L l</p> <p>M m</p> <p>N n</p> <p>O o</p> <p>P p</p> <p>Q q</p> <p>R r</p> <p>S s</p> <p>T t</p> <p>U u</p> <p>V v</p> <p>W w</p> <p>X x</p> <p>Y y</p> <p>Z z</p>	<p>identifiers</p> <p>- <i>minus, vector</i></p> <p>* <i>VECTOR CROSS PRODUCT</i></p> <p>.</p> <p>/</p> <p> <i>operators</i> <i>CR and II for concatenation</i></p> <p>& <i>LOGICAL AND</i></p> <p>=</p> <p><</p> <p>></p> <p>#</p> <p>@ <i>SUBSCRIPTING</i></p> <p>\$</p> <p>,</p> <p>;</p> <p>:</p> <p>(blank) <7</p> <p>(</p> <p>) <i>delimiters</i></p> <p>.</p> <p>% <i>keywords</i></p>
<p>0</p> <p>1</p> <p>2</p> <p>3</p> <p>4</p> <p>5</p> <p>6</p> <p>7</p> <p>8</p> <p>9</p>	<p>[</p> <p>]</p> <p>{</p> <p>}</p> <p>!</p> <p>?</p> <p>~ <i>iteration</i></p> <p>"</p>

identifiers
reserved words
literals

identifiers
literals

(macros)

2.3 HAL/S Primitives

HAL/S syntax diagrams ultimately express all syntactical elements in terms of a small number of undefined 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 only one or two exceptions they are constructed from alphabetic characters alone. Reserved words fall into two categories, keywords, and built-in function names. In the syntax diagrams, and in the accompanying text, reserved words are indicated by upper case characters. A list of keywords is given in Appendix A., and of built-in function names in Appendix B.

2.3.2 Identifiers

An identifier is a name assigned by the programmer to be a data item, label, or other entity. Before its attributes are specified, 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 first appearance 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>

<structure template>

where § ~ { arithmetic
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.

FORMATION RULES (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:

‡ ddddddd.dddddddd <exponents> d = decimal digit

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

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

$$B \langle \text{power} \rangle = 2^{\langle \text{power} \rangle}$$

$$E \langle \text{power} \rangle = 10^{\langle \text{power} \rangle}$$

$$H \langle \text{power} \rangle = 16^{\langle \text{power} \rangle}$$

where <power> is a signed integer number.

EXAMPLES:

0.123E16B-3

45.9

-4

FORMATION RULES (bit literals)

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

BIN	<repetition>	'bbbbbb'	b = binary digit
OCT	<repetition>	'oooooo'	o = octal digit
HEX	<repetition>	'hhhhhh'	h = hexadecimal digit
DEC	<repetition>	'dddddd'	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.

restricted to one word only

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'

FORMATION RULES (character literals)

always changing in HAL/S

1. Literals of character type are denoted syntactically by <char literal>.
2. The form of a character literal is:

'cccccccccccc'

where c is any character in the HAL/S extended character set.

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 character is denoted by an apostrophe pair, (i.e. the representation of "dog's" would be 'DOG''S', for example).
4. The character pair /* is always taken to be the opening ~~delimiter of a comment even in a character literal.~~ *Turned into a line*

EXAMPLES:

''

'ONE TWO THREE'

'DON''T'

2.4 Single Line and Multiple Line Source Text

In preparing the source text of HAL code, single or multiple line format may optionally 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, and subscripts are written below respectively, the line where the operands they refer to are written. Of the two formats, the 2-dimensional is regarded as standard, since it follows usual mathematical practice.

RULES FOR EXPONENTS:

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

EXAMPLES:

$$A^I \rightarrow A^{**I}$$

$$A^{I^J} \rightarrow A^{**I^{**J}}$$

2. Operations are evaluated right to left (see Section 6.1.1).
3. If an exponent is subscripted, its subscript must be given its 1-dimensional description.

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



return from subscript

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

2. In the 1-dimensional form of the HAL/S subscript, the subscript expression is delimited 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 unsubscripted <arith var>

EXAMPLE:

$$A_J \rightarrow A\$J$$

4. IF a subscript expression contains an exponentiation operation, the latter must be given its 1-dimensional representation.

2.5 Other Aspects of the Source Text

Any HAL source text consists of sequences of HAL/S primitives of the types described. It is obviously of great importance for a compiler to be able to tell the end of one primitive 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. Generally blanks are required as separators between identifiers, keywords, and literals. Except in character literals, consecutive blanks are syntactically equivalent to a single blank.

Comments may be imbedded within HAL source text wherever blanks are legal. A comment is delimited at the beginning 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 /).

3.0 HAL/S BLOCK STRUCTURE AND ORGANIZATION

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

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

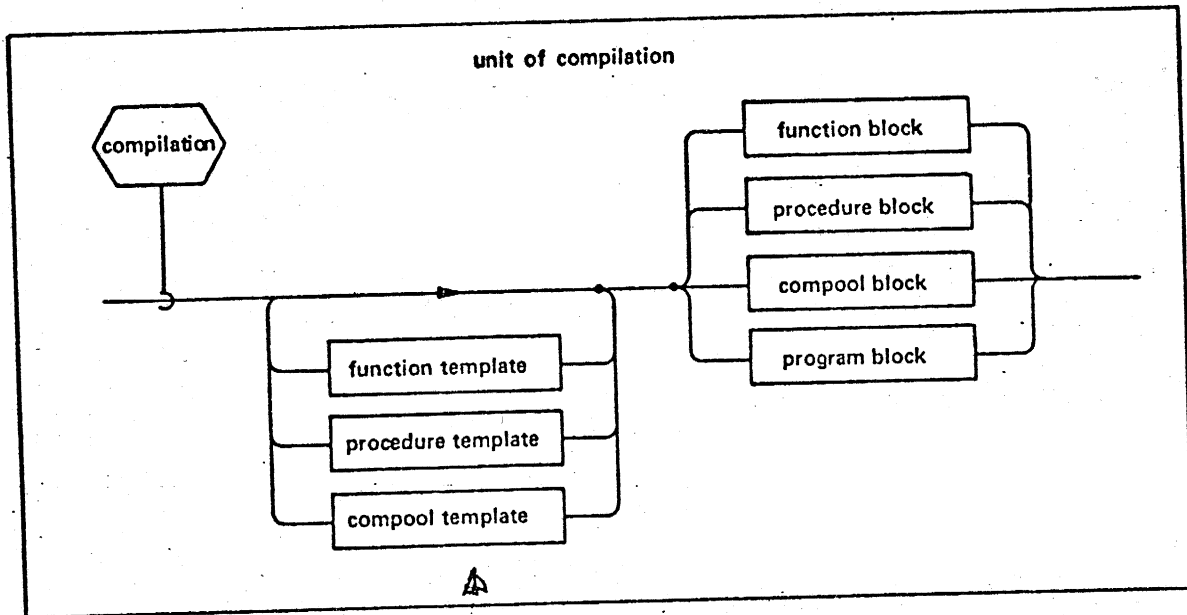
- program modules - characterized by being independently executable.
- external procedure and function modules - characterized by being callable from other modules.
- compool modules - forming common data pools for the program complex.

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

3.1 The Unit of Compilation

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

SYNTAX:

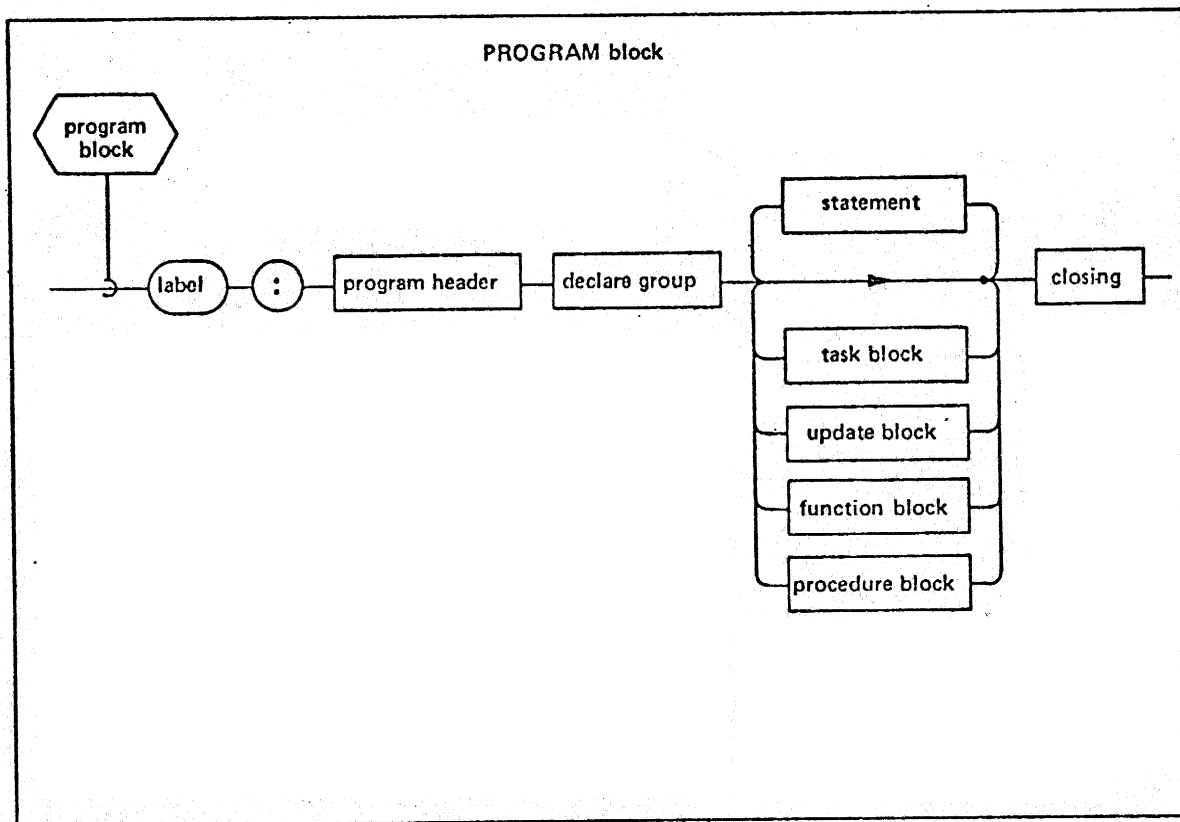


produced automatically by prior compilation and stored in intermediate file.

3.2 The PROGRAM Block

The PROGRAM block delimits a main, independent body of HAL/S code consisting of a <declare group> and any number of executable <statement>s and/or nested PROCEDURE, FUNCTION, TASK, and UPDATE blocks. Delimiting is done by a <program header> and a <closing>.

SYNTAX:



EXAMPLE:

SAMPLE: PROGRAM;

DECLARE A SCALAR;

DECLARE B VECTOR;

⋮

BETA: FUNCTION(Y);

⋮

CLOSE BETA;

ALPHA: PROCEDURE

⋮

CLOSE ALPHA;

A = K + BETA(X);

⋮

B = R*V;

⋮

CALL ALPHA;

CLOSE SAMPLE;

] declare group

] function block

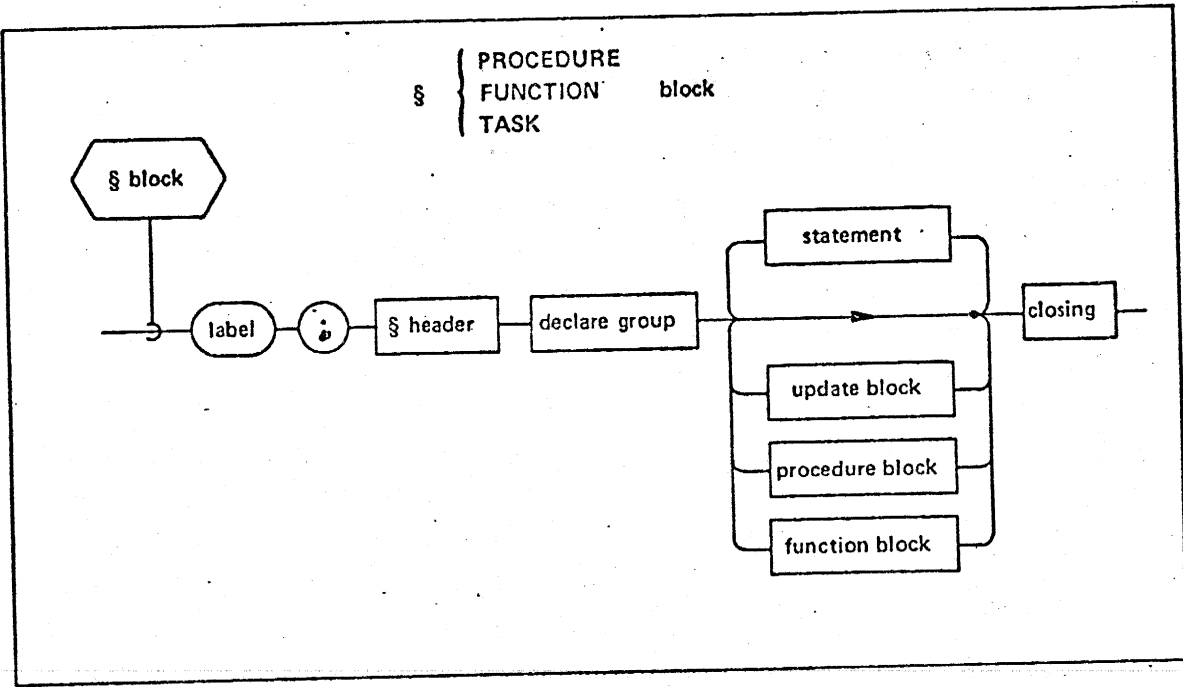
] procedure block

] executable stmts.

3.3 The PROCEDURE, FUNCTION and TASK Blocks

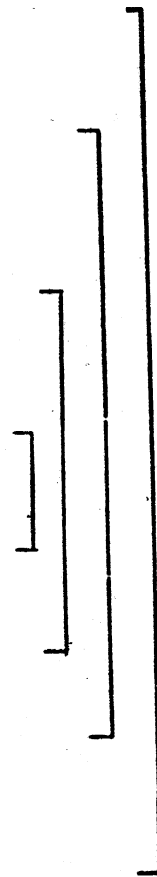
PROCEDURE, FUNCTION, and TASK blocks share a common purpose in serving to structure HAL/S code into an interlocking modular form. The major semantic distinction between the three types of blocks is the manner of their invocation (described in Section 7.4, 6.4, and 8.3 respectively). Each block is delimited by a header statement of the proper type and a <closing>. The blocks consist of a <declare group> to declare data local to the block, followed by any number of executable <statement>s and/or nested PROCEDURE, FUNCTION, and UPDATE blocks.

SYNTAX:



EXAMPLE OF NESTING PROCEDURES AND FUNCTIONS:

```
NEST: PROCEDURE;  
  DECLARE A VECTOR;  
  ⋮  
  ALPHA: PROCEDURE;  
    DECLARE B;  
    ⋮  
    BETA: FUNCTION(X);  
      DECLARE X;  
      ⋮  
      GAMMA: PROCEDURE;  
        ⋮  
        CLOSE GAMMA;  
        ⋮  
      CLOSE BETA;  
      ⋮  
    CLOSE ALPHA;  
    ⋮  
  CLOSE NEST;
```

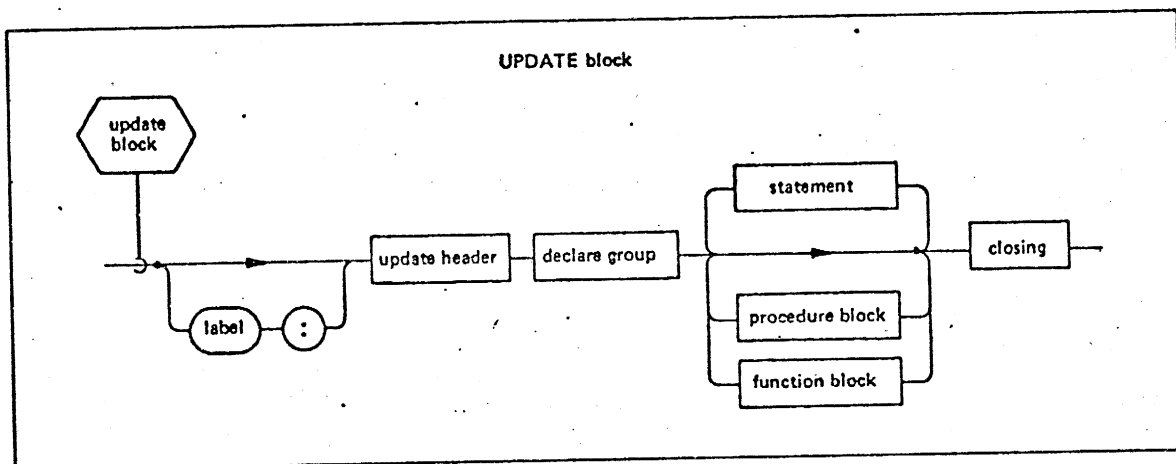


The diagram consists of three vertical brackets on the right side of the code, indicating the nesting levels. The innermost bracket starts at the 'GAMMA: PROCEDURE;' line and ends at the 'CLOSE GAMMA;' line. The middle bracket starts at the 'BETA: FUNCTION(X);' line and ends at the 'CLOSE BETA;' line. The outermost bracket starts at the 'ALPHA: PROCEDURE;' line and ends at the 'CLOSE ALPHA;' line. There is also a long vertical line on the far right that spans from the 'NEST: PROCEDURE;' line down to the 'CLOSE NEST;' line, representing the outermost nesting level.

3.4 The UPDATE Block

The UPDATE block is used to control the sharing of data by more than one real time process (see Section 7.) and is invoked when it is encountered in the normal flow of execution. The UPDATE block is delimited by an <update header> and a <closing>. The block consists of a <declare group> to declare data local to the UPDATE block, followed by any number of executable <statement>s (except I/O and real-time statements) and/or nested PROCEDURE and FUNCTION blocks.

SYNTAX:



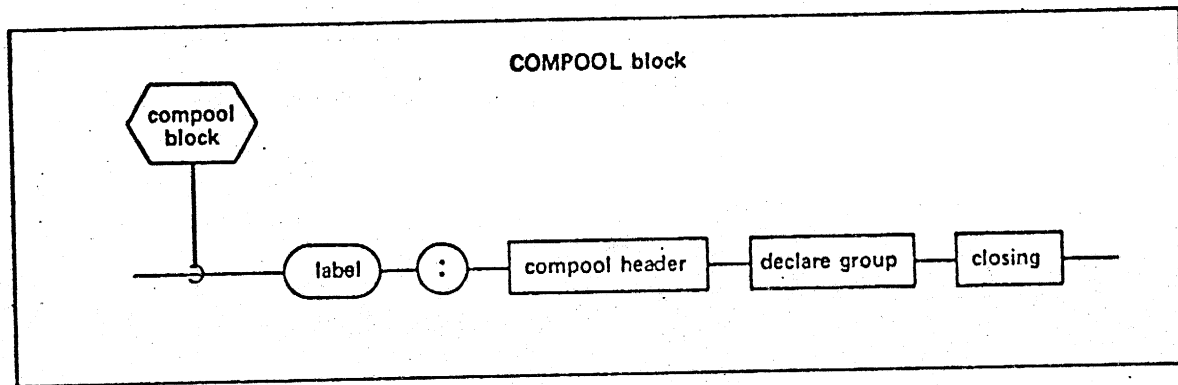
EXAMPLE:

```
A: TASK;
  :
  :
  UPDATE;
  :
  :
  M=N+P;
  :
  :
  CLOSE;          /* END OF UPDATE BLOCK */
  :
  :
  CLOSE A;
```


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. The number of COMPOOL blocks allowed in a program complex is implementation dependent.

SYNTAX:

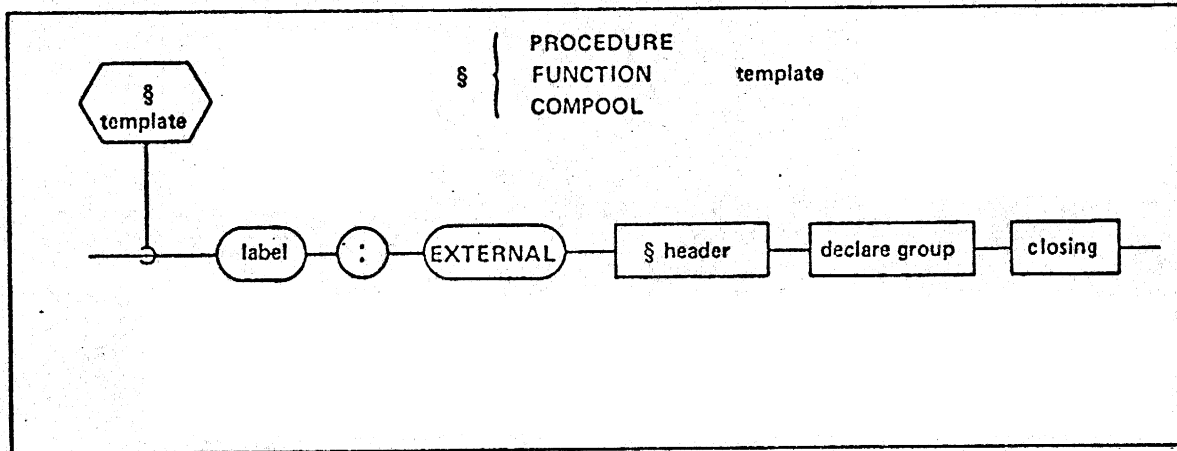


EXAMPLE: MAIN_COMPOOL: COMPOOL;
 DECLARE M MATRIX;
 DECLARE V VECTOR INITIAL(1,0,0)] declare
 : group
 :
 CLOSE MAIN_COMPOOL;

3.6 PROCEDURE, FUNCTION, and COMPOOL Templates

Block templates are used to provide the outermost code block of a <compilation> with information concerning external code blocks. Both the <label> and the header statement must be identical to those of the corresponding code block, except the keyword EXTERNAL on the leftmost side of the header statement distinguishes it from an otherwise identical code block. A COMPOOL template declares a common data pool identical to that of the corresponding COMPOOL block; a PROCEDURE or FUNCTION template declares the formal parameters of the corresponding PROCEDURE or FUNCTION block. Depending upon implementation, the compiler system may generate and maintain templates automatically.

SYNTAX:



EXAMPLES:

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

```
BUZZ: EXTERNAL FUNCTION([X]);  
    DECLARE X ARRAY(4) VECTOR;  
CLOSE BUZZ;
```

```
BAKER: PROCEDURE(A) ASSIGN(B)  
    DECLARE VECTOR(6), A, B, C;  
    ⋮  
    A = B + C;  
CLOSE BAKER;
```

procedure
block

```
BAKER: EXTERNAL PROCEDURE(A) ASSIGN(B);  
    DECLARE VECTOR(6), A, B; /* NOTE ONLY ARGUMENTS ARE DECLARED */  
CLOSE BAKER;
```

procedure
template

```
ABLE: PROGRAM;  
    ⋮  
    CALL BAKER(ZETA) ASSIGN(PHI);  
    ⋮  
CLOSE ABLE;
```

program
block

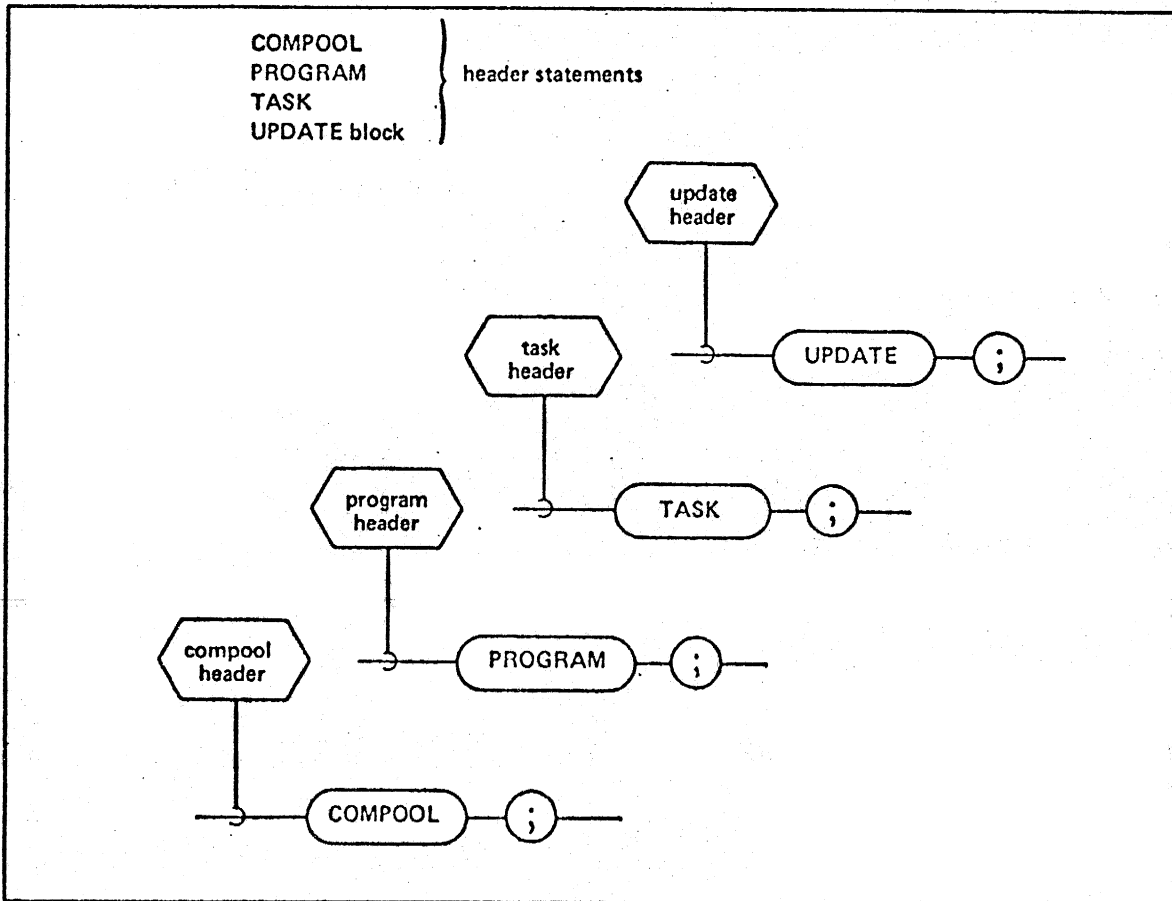
3.7 Block Delimiting Statements

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, and the <closing> statement is the last to be executed before exit. A COMPOOL block, containing only declarations of data, is not executable.

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.

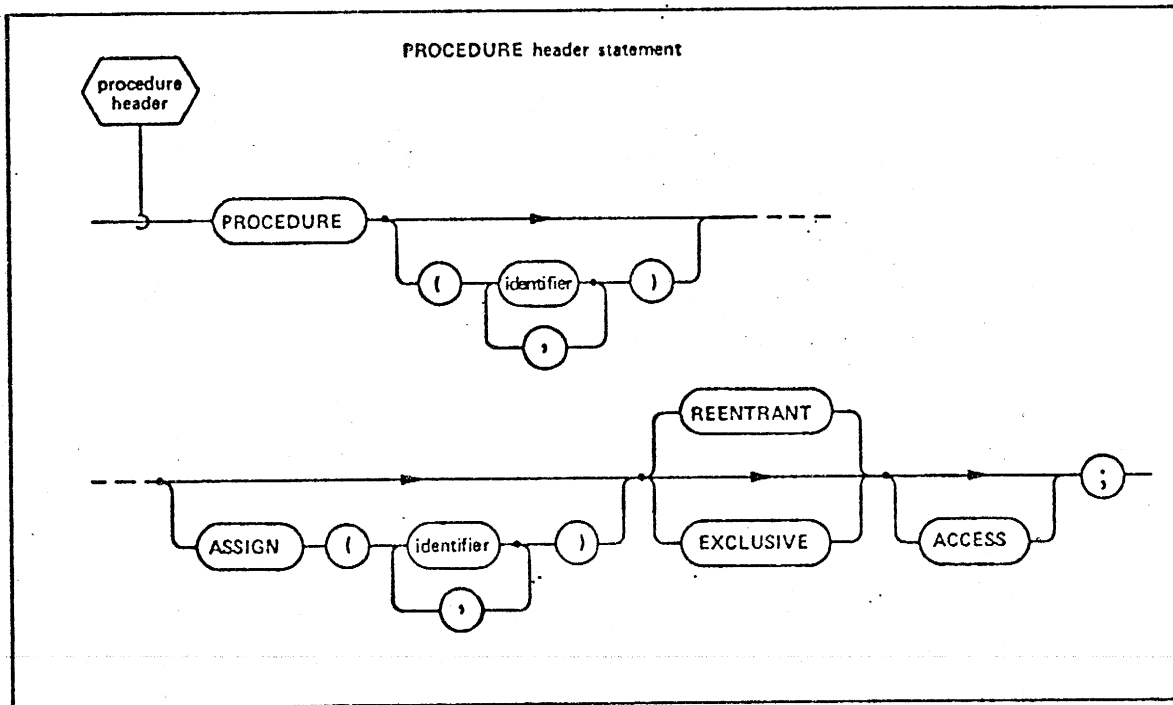
SYNTAX:



3.7.2 The Procedure Header Statement

The procedure header delimits the start of a PROCEDURE block or PROCEDURE template. The <identifiers> following the PROCEDURE keyword are "input parameters" whose values may not be changed within the code block; the <identifiers> following the ASSIGN keyword are "assign parameters" whose values may be altered within the code block. All of these parameters must have data declarations in the <declare group> of the PROCEDURE block or template. The keyword REENTRANT allows real-time sharing of the PROCEDURE block. The keyword EXCLUSIVE allows only one real-time process to use the PROCEDURE block at a given time; any other processes must wait to use the PROCEDURE block until the first is finished executing it. The keyword ACCESS places implementation dependent managerial restrictions on which <compilation>s may reference an external PROCEDURE block.

SYNTAX:

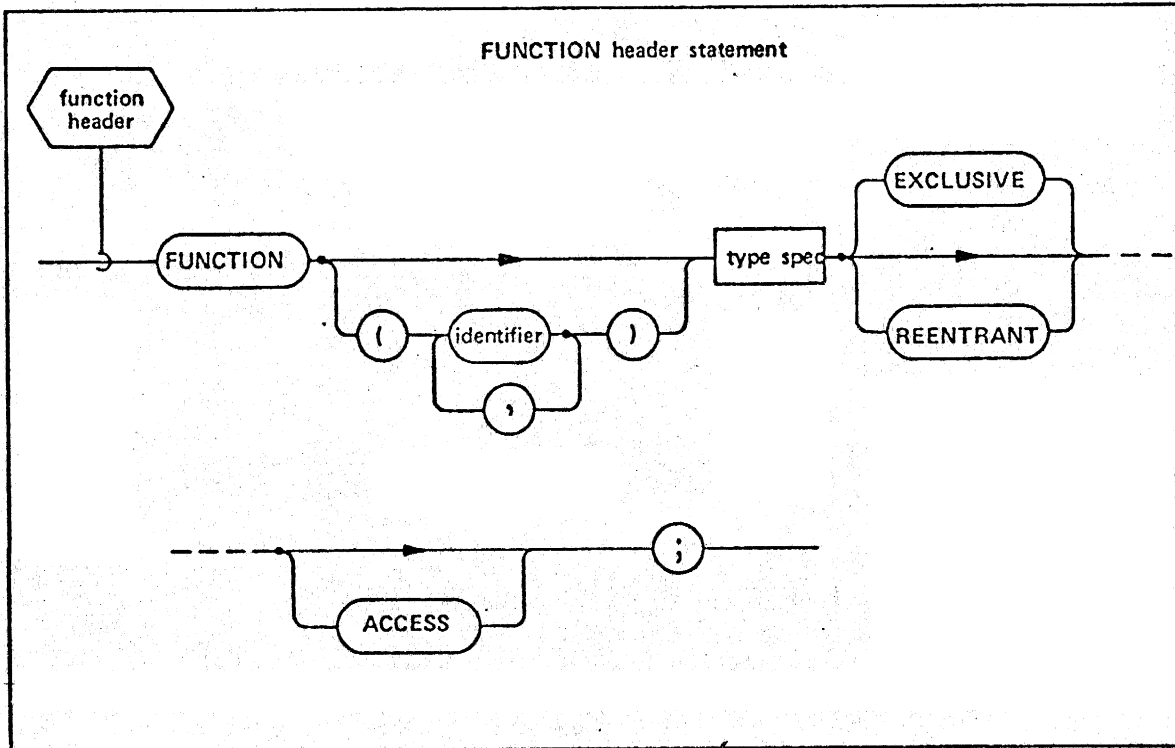


EXAMPLES: PROCEDURE ASSIGN(B);
 PROCEDURE (\bar{V}, M^*) ASSIGN(\bar{N}) EXCLUSIVE;
 PROCEDURE(X) ACCESS;

3.7.3 The Function Header Statement

The function header delimits the start of a FUNCTION block or FUNCTION template. The <identifiers> following the FUNCTION keyword are "input parameters" whose values may not be changed within the code block, and whose data type is declared in the <declare group> of the FUNCTION block or template. <type spec> identifies the type of value returned by the FUNCTION block (<type spec> may not be an event type). The keyword REENTRANT allows real-time sharing of the FUNCTION block. The keyword EXCLUSIVE allows only one real-time process to use the FUNCTION at a given time; any other process must wait to use the FUNCTION block. The keyword ACCESS places implementation dependent managerial restrictions on which <compilation>s may reference an external FUNCTION block.

SYNTAX:



EXAMPLES:

```

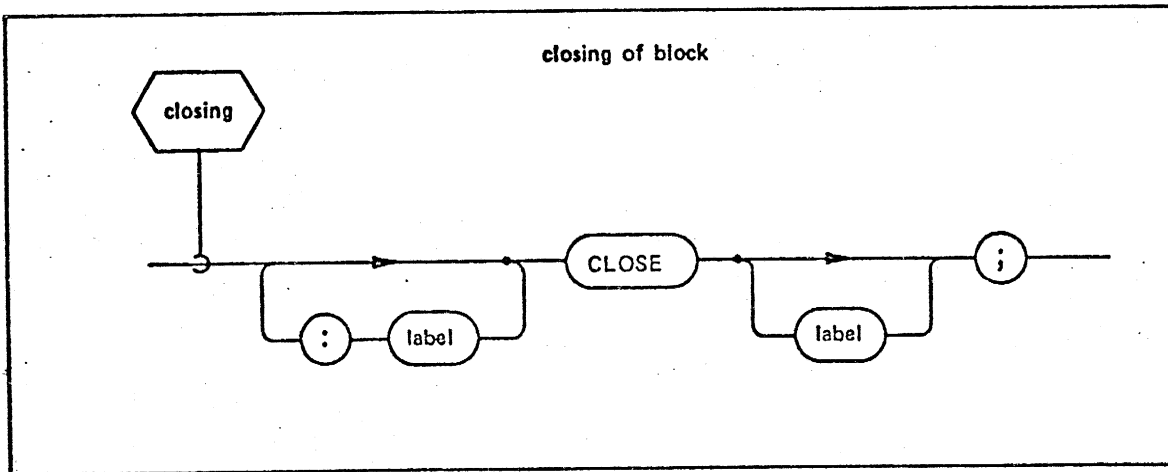
FUNCTION(*A) SCALAR REENTRANT;
FUNCTION (ALPHA,BETA) VECTOR;

```

3.7.4 The CLOSE Statement

For all code blocks and block templates, the CLOSE statement is the <closing> delimiter. If the CLOSE keyword is followed by a <label>, the <label> must be the name of the block. The <closing>s of the COMPOOL blocks and block templates cannot have a <label> to the left of the keyword CLOSE.

SYNTAX:



EXAMPLES:

ALL_DONE: CLOSE;

CLOSE MAJOR_COMPOOL;

*TD here
1st day*

3.8 Name Scope Rules

As a consequence of the code block structure of HAL/S, the scope of a name (<identifier>), i.e. a <variable name> or <label> is defined as the block in which it is declared and potentially extends to all contained and nested blocks. The scope of a name is therefore the region in which it is potentially recognizable. For example, names defined in a <compool block> are potentially recognized throughout every compilation unit; i.e. <program block>s and external procedures and functions; names defined in a <program block> may be recognized in all enclosed <task block>s, <procedure block>s, <function block>s, or <update block>s, etc. Duplicate names are allowed in different blocks where the outer declaration of the name is superseded, in the inner block only, by the explicit declaration. A name defined only within an inner block is never recognized in an outer block.

HAL/S does not permit GO TO's between blocks of code, thus a branch from an inner block to an outer block is specifically disallowed.

EXAMPLE:

```
outer name scope [ ALPHA: PROGRAM;
                  DECLARE X;      /* X IS KNOWN EVERYWHERE */
                  DECLARE Y;      /* Y IS KNOWN ONLY OUTSIDE BETA */
                  ⋮
inner name scope [ BETA: PROCEDURE; /* LABEL BETA KNOWN IN ALPHA */
                  DECLARE Y;      /* NEW Y KNOWN ONLY IN BETA */
                  DECLARE Z;      /* Z KNOWN ONLY IN BETA */
                  ⋮
                  CLOSE BETA;
                  ⋮
                  DELTA: Y=0;     /* DELTA NOT KNOWN IN BETA */
                  CALL BETA;      /* BETA CAN BE CALLED ONLY FROM ALPHA */
                  ⋮
                  CLOSE ALPHA;
```

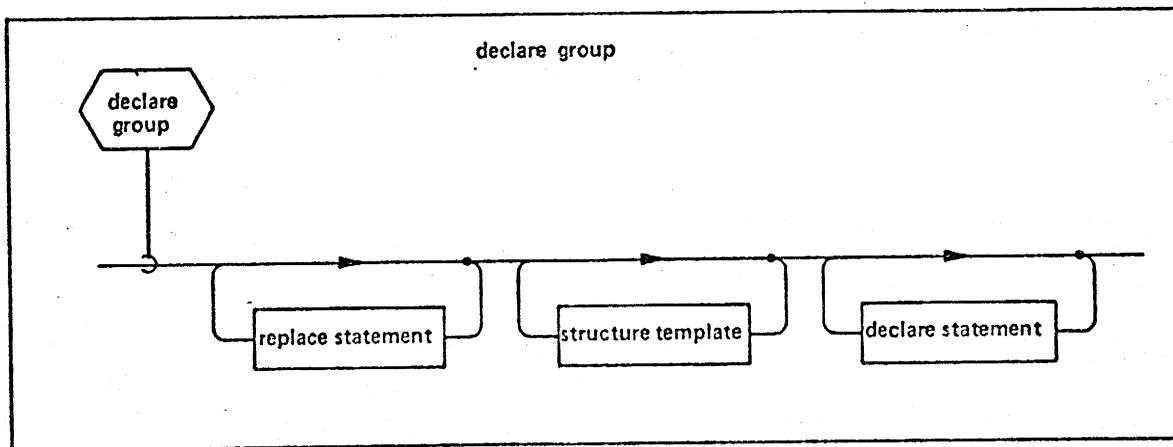

4.0 DATA AND LABEL DECLARATIONS

The HAL/S language possesses a comprehensive set of data types for use in both applications and systems programming situations. To encourage clarity and decrease the frequency of errors of omission, all data is required to be defined in specific areas of a HAL/S compilation called "declare groups".

4.1 The Declare Group

A <declare group> is a collection of data and label declarations possibly consisting of <replace statement>s, <structure template>s, and <declare statement>s.

SYNTAX:



EXAMPLES: REPLACE PI BY '3.14159';
 REPLACE MU BY '1234';

STRUCTURE A:
 1 B SCALAR,
 1 C INTEGER;

} Replace group

} Structure Template

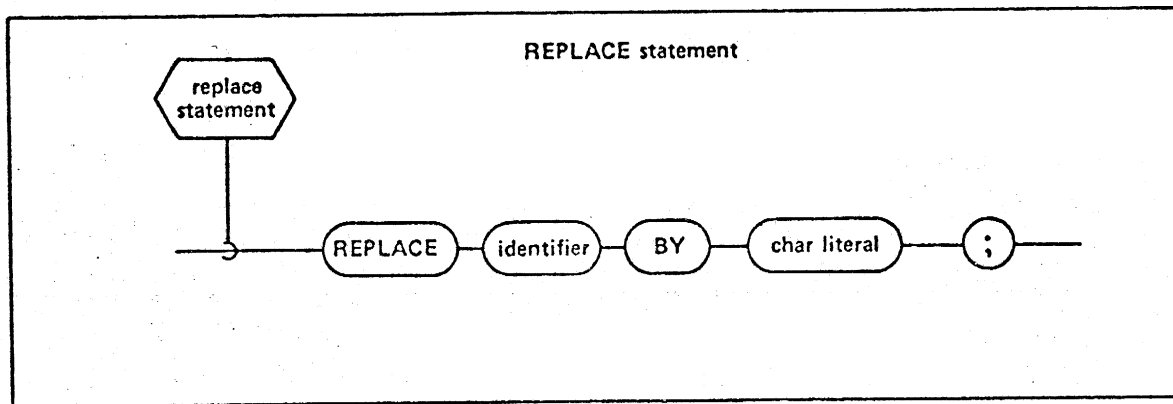
```
DECLARE A A_STRUCTURE;  
DECLARE INTEGER, M, N;  
DECLARE  $\bar{V}$  VECTOR;
```

Declare Group

4.2 The REPLACE Statement

The REPLACE statement is used to define a name (i.e. <identifier>) as a text substitution. Any HAL/S code containing reference to the <identifier> is treated as if the text of <char literal> had instead appeared in that position. <identifier> may not be a formal parameter in a <procedure header> or <function header>, nor may an <identifier> in a REPLACE statement be the subject of a replacement itself.

SYNTAX:



EXAMPLES:

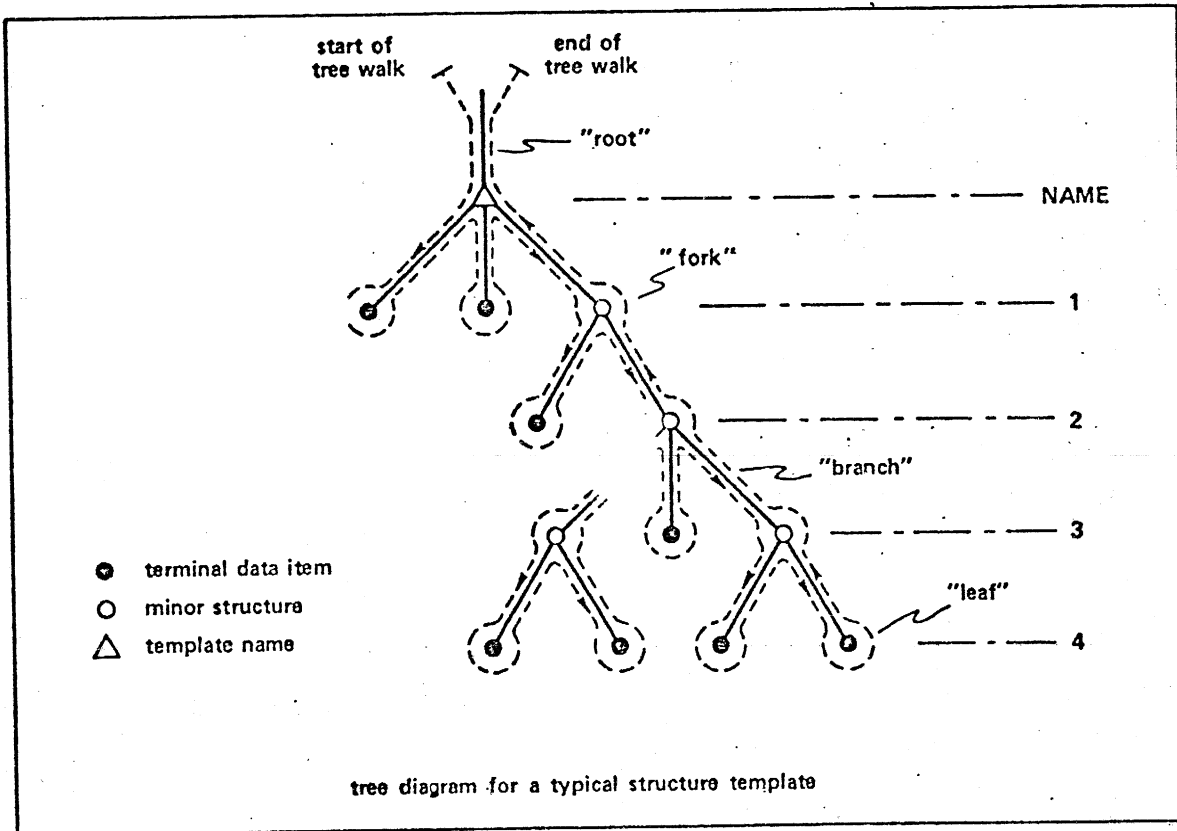
REPLACE ALPHA BY 'J+1';

REPLACE TERMINATION BY 'GO TO FINISH';

4.3 The Structure Template

In HAL/S, a "structure" is a hierarchical organization of generally inhomogeneous data items. Conceptually the form of the organization is a "tree", with a "root", "branches", and with the data items 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 structure data having that organization. The tree organization is defined by a <structure template>.

The following figure shows a typical tree organization in its conceptual form:

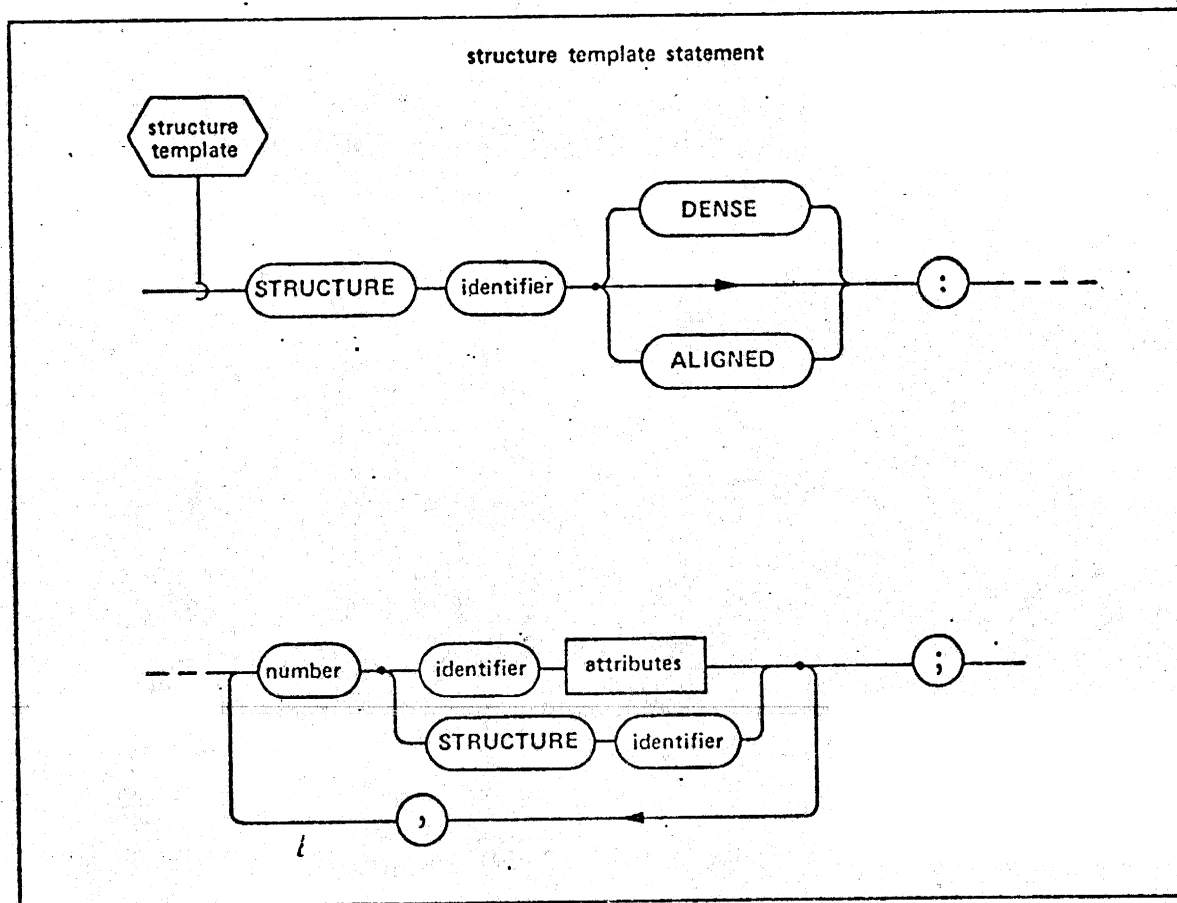


The keywords DENSE and ALIGNED denote data packing attributes of all structures possessing the <structure template> as explained in Section 4.6.

The names of minor structures (i.e. each fork or diagram) and terminal data items must be defined in the same order as the tree walk (shown on diagram) passes them on the left (see example below which shows this in relation to the above diagram).

The form STRUCTURE identifier appearing after the colon causes a previously defined <structure template> called <identifier> to be incorporated as part of the <structure template> being defined.

SYNTAX:



EXAMPLE (corresponding to tree diagram shown on previous page):

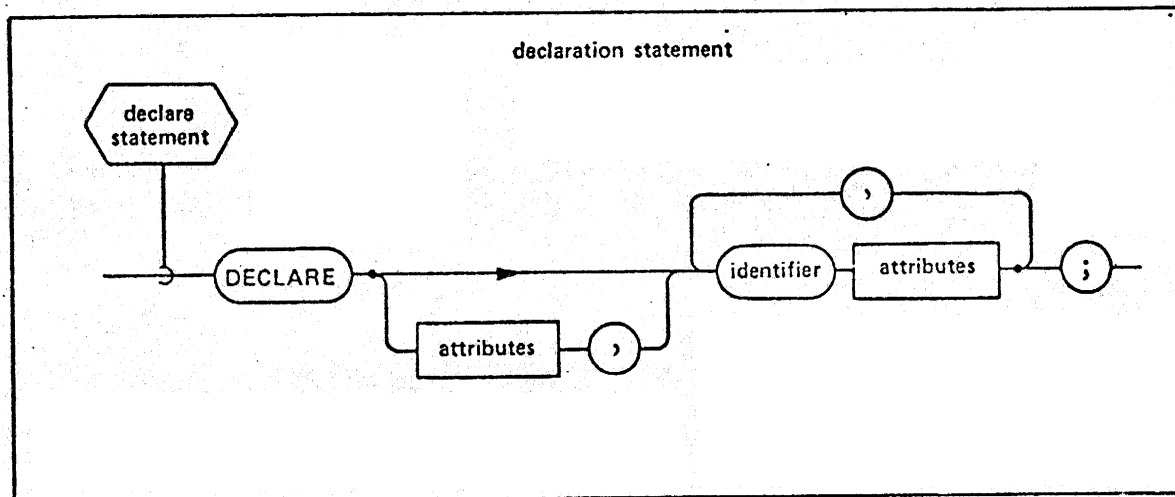
STRUCTURE OMEGA DENSE:

```
1 PHI ARRAY(50) BIT(31),
1 ZETA SCALAR,
1 ALPHA,
  2 BETA ARRAY(25),
  2 GAMMA,
    3 LAMBDA_1,
      4 MHOS INTEGER,
      4 COND SCALAR,
    3 NU,
    3 LAMBDA_2,
      4 OHMS INTEGER,
      4 RESIS SCALAR;
```

4.4 The DECLARE Statement

The DECLARE statement is used to declare variable names, and labels, and to define their characteristics, or <attributes>. Any <inflections> given immediately after the keyword DECLARE are characteristics (factored <attributes >) of all <identifier>s in the DECLARE statement. Each <identifier> and associated <attributes > constitutes the declaration of the particular <identifier>, and must not conflict with any factored <attributes >. The appearance of either a label or a variable name determines the form of the <attributes > (see Sections 4.5 and 4.6 respectively).

SYNTAX:



EXAMPLES: DECLARE INTEGER, A,B, ARRAY(5);
 DECLARE M ARRAY(10) MATRIX(2,3);
 DECLARE ABL FUNCTION SCALAR;

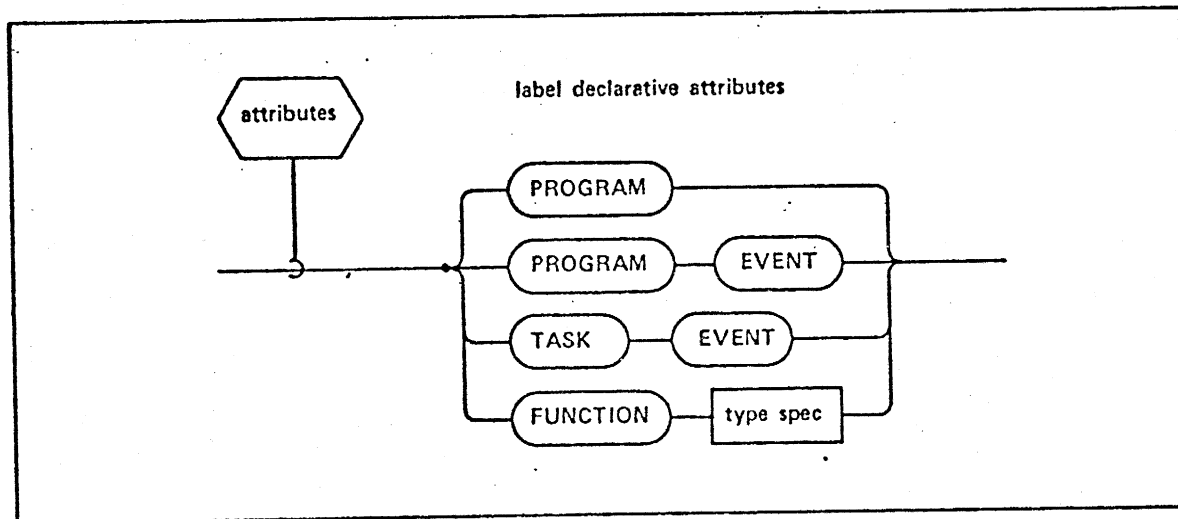
4.5 Label Declarative Inflections

Label declarations in HAL/S are used to define the names of PROGRAM, TASK and FUNCTION code blocks. The forms PROGRAM and PROGRAM EVENT may only appear in the <declare group> of a <compool block> and its corresponding template to allow any external <program block> to be referenced by a <compilation>. The keyword EVENT allows a process-event (see Section 8.9) to be attached to the <program block>.

The form TASK EVENT may only appear in the <declare group> of a <program block> to allow the named <task block> to have attached to it an identically named process-event.

The form FUNCTION <type spec> is used to define the name and type of a <function block>. The function defined this way must have at least one formal parameter, none of which may be arrayed. A function declaration is required whenever a function is used prior to the appearance of its code block.

SYNTAX:



EXAMPLES: DECLARE ALPHA PROGRAM;
 DECLARE USER_FUNC FUNCTION INTEGER;
 DECLARE BETA TASK EVENT;

4.6 Data Declaration Attributes

Data declaration attributes are used to define an <identifier> to be a variable name or part of a structure template, and to describe its characteristics. If <attributes> appears in a <declare statement>, it defines a variable name. If <attributes> appears in a <structure template> it defines either a minor structure, or a terminal data item of the template. Terminal data items have very similar properties to variable names.

The keyword ARRAY allows the specification of the number and sizes of the dimensions in the array. Each <arith exp> denotes the integral size of a dimension, while an asterisk denotes a linear array of unknown length which is used as a formal parameter of a procedure or function. The actual length is that of the corresponding argument on invocation.

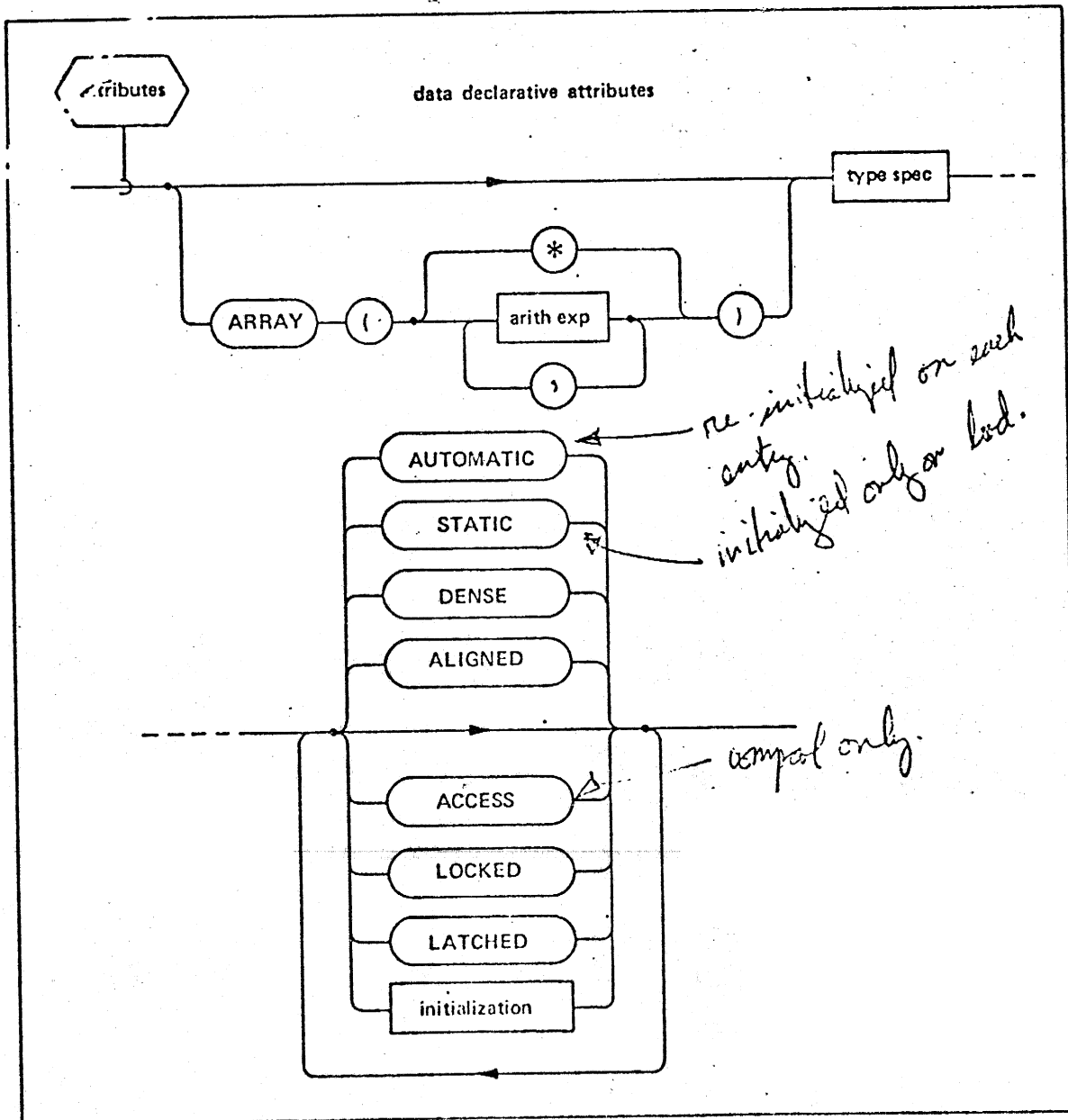
The following attributes are allowed for variable names:

- AUTOMATIC/STATIC - an <identifier> with the AUTOMATIC attribute is initialized upon every entry into the code block containing its declaration. An <identifier> with the STATIC attribute is initialized once upon first entry into the code block. Generally if neither keyword appears STATIC is assumed.
- DENSE/ALIGNED - If the <identifier> has the ALIGNED attribute, its storage is arranged on natural word or fractional word boundaries so as to optimize speed of reference. If the <identifier> has the DENSE attribute its storage is packed so as to minimize the size of storage area required. In the absence of either keyword, ALIGNED is assumed.
- ACCESS - causes managerial restrictions to be placed upon the usage of the <identifier> as a variable in assignment contexts, and may only be used in the <declare group> of a <compool block> or its template.

- LOCKED - may only be used in the <declare group> of a <compool block> or its template and causes use of the <identifier> to be restricted to UPDATE blocks (see Section 8.10).
- LATCHED - only applies to event variables as specified in Section 4.7.
- <initialization> - allows initialization of an <identifier> as specified in Section 4.8.

Terminal data items and minor structures may only use the attributes DENSE or ALIGNED.

SYNTAX:



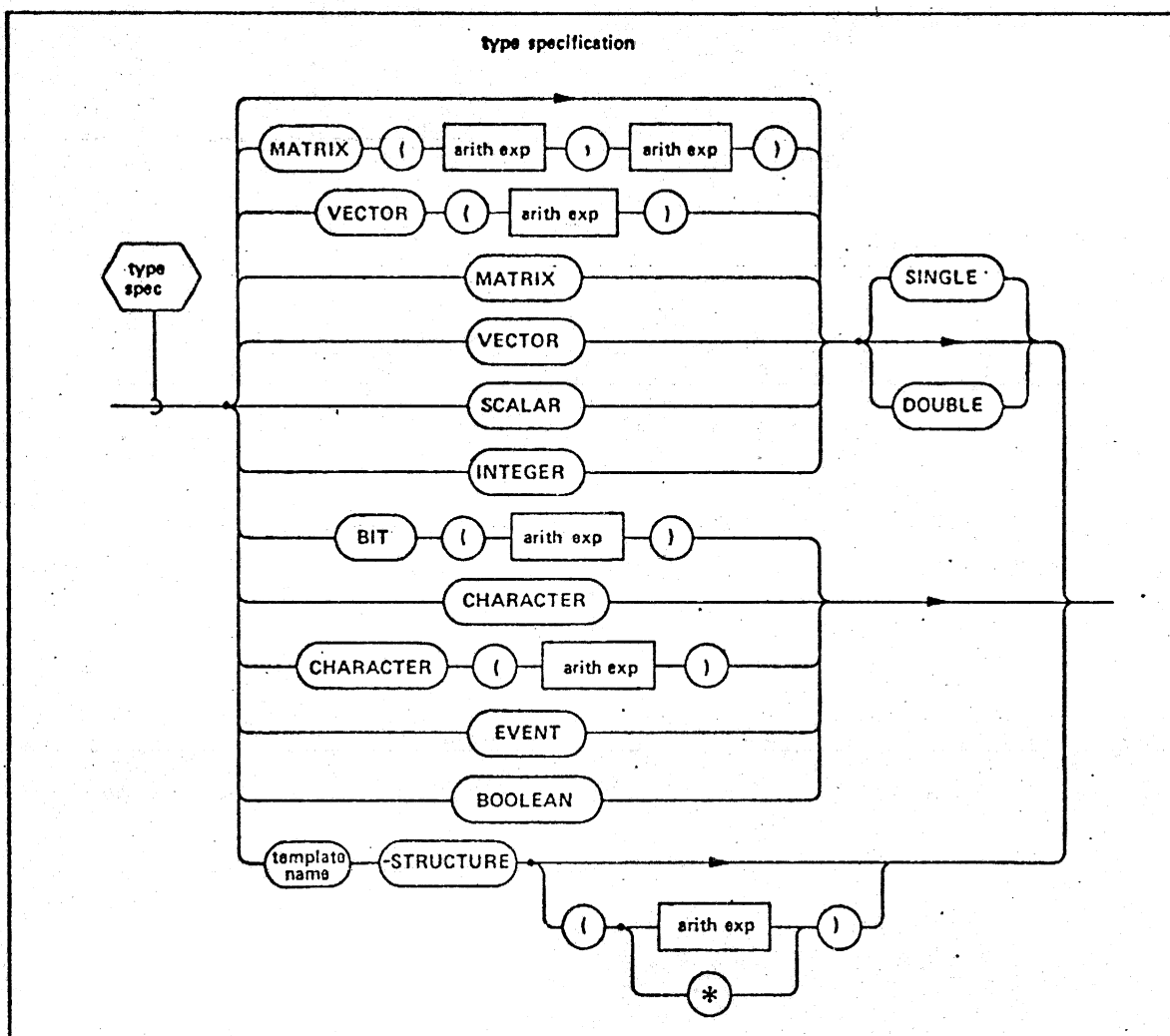
4.7 Type Specification

The type specification or <type spec> provides a means of defining the type (and precision of VECTOR, MATRIX, INTEGER, and SCALAR type only) of variable names and terminal data items of structure templates. If there is no <type spec> given, then the implied type of a variable name or terminal data item is SCALAR with SINGLE precision; if <type spec> consists only of the keyword SINGLE or DOUBLE then it is SCALAR of the indicated precision.

The <arith exp> of a VECTOR is its length; the default value is 3. Similarly, the two <arith exp>s of a MATRIX are its row and column dimensions respectively; the default is a 3-by-3 matrix. The <arith exp> of a CHARACTER type denotes its maximum length whose default value is 8. BIT(<arith exp>) indicates a bit type of the specified length. Both BOOLEAN and EVENT indicate a bit type of 1-bit length, however, EVENT is used in real time programming situations (see Section 8.8).

The phrase <identifier>-STRUCTURE denotes structure type with a tree organization given by a previously defined template named <identifier>. If the structure variable name in the declare statement is the same as the <template name>, then the structure is said to be unqualified; if they differ then the structure is said to be qualified (see Section 5.2). <arith exp> gives the number of copies of the structure. The copy specification may only be an asterisk if the structure is a formal parameter of a procedure or function, in which case the actual number of copies is supplied by the corresponding argument on invocation of the procedure or function.

SYNTAX:



EXAMPLES: MATRIX(2,2) DOUBLE
 Z-STRUCTURE(15)
 CHARACTER(7)

4.8 Initialization

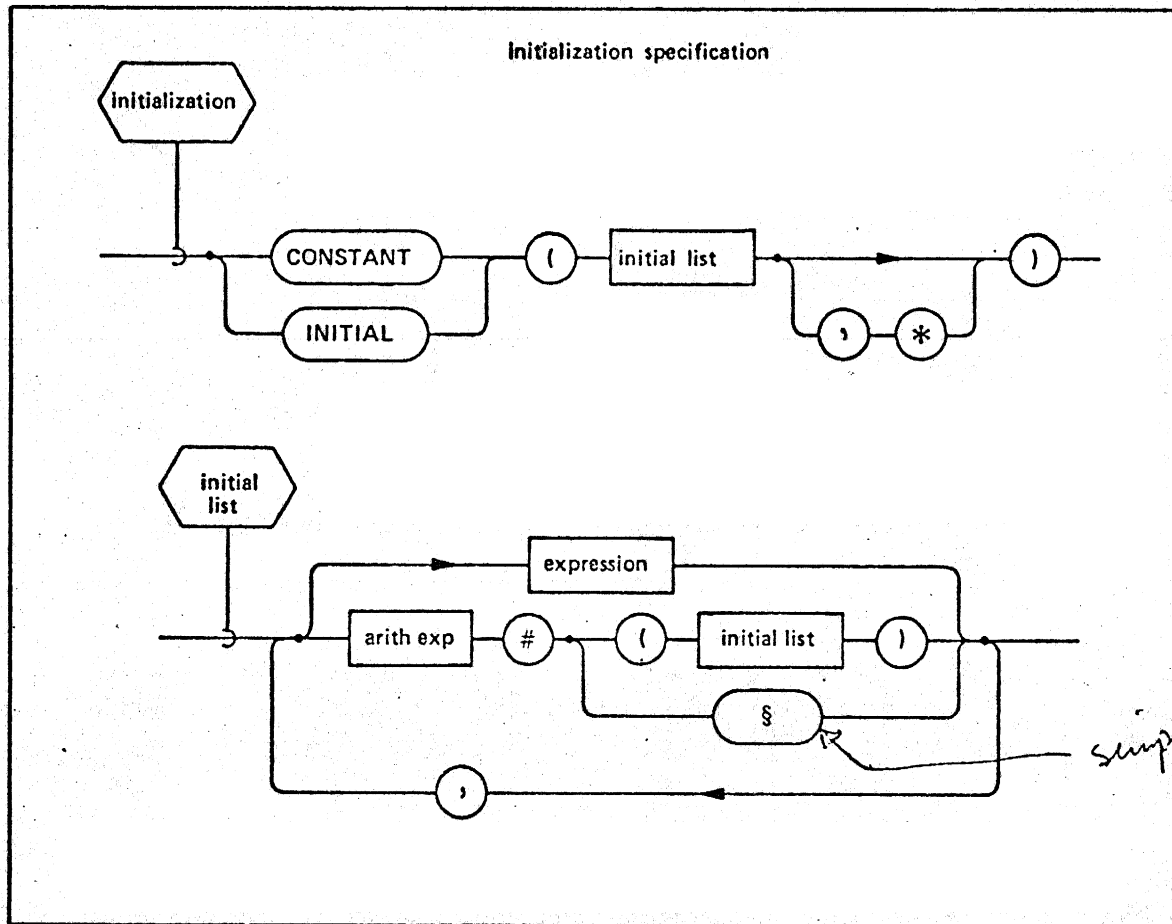
The <initialization> starts with the keyword INITIAL or CONSTANT. A CONSTANT <initialization> makes it illegal for <identifiers> to appear in an assignment context since its value may never be changed.

A simple <initial list> is a sequence of one or more <expression>s of the proper type which are computable at compile-time. A simple <initial list> may be repeated to form a more complex <initial list> by the phrase <arith exp>#. <arith exp># may also precede a single literal or a single unsubscripted variable name, (denoted by \$ in the syntax diagram).

In general, the number of values in the <initial list> must be equal to the total number of components of the variable. However, an asterisk following the <initial list> implies the partial initialization of a variable name.

If the variable has array specification, and is an integer or a scalar, a single value in the <initial list> may be used to specify the initial value of all the array elements. Similarly, for vector, matrix, bit or string initialization a single value in the <initial list> can specify the initial value of each individual component, or of each component of an array of vector, matrix, bit or character type. If the variable is an array of vectors or matrices, and the number of values in the <initial list> is equal to the number of components of the vector or matrix, then those values are applied to all array elements alike. If the variable is a structure with multiple copies, and the number of values in the <initial list> is exactly equal to the total number of data elements in one copy of the structure, then each structure copy is identically initialized with those values.

SYNTAX:



EXAMPLES:

```

DECLARE A ARRAY(8) INTEGER INITIAL(2#(1,3#5));
DECLARE B ARRAY(5) BIT(7) CONSTANT(5#(BIN'1010011'));
DECLARE C CHARACTER(5) INITIAL('ALPHA');
DECLARE IDENTITY_MAT MATRIX INITIAL(1,0,0,0,1,0,0,0,1);
DECLARE V ARRAY(4) VECTOR(5) INITIAL(1,2,3,4,5);
    
```

right most index in an array is varying fastest.

(2,2,2)

*111
112
121
122*

*211
212
221
222*

natural sequence

5.0 DATA REFERENCING CONSIDERATIONS

Central to the HAL/S language is the ability to access and change the values of variables. Section 4. dealt comprehensively with the way in which variable 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

A "simple variable" is any variable which is 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. References to the whole 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).

5.2.1 Unqualified Structures

If a structure is unqualified, then any part of it, either minor structure or terminal, may be referenced by using the name of the part as it appears in the <structure template> definition. If a minor structure is referenced, the name becomes syntactically a <structure var name>. If a terminal is referenced, then syntactically the name becomes

a <\$var name>, where \$ stands for any of the types, arithmetic, bit, character, or event, as appropriate to the attributes of its definition in the template.

EXAMPLE:

STRUCTURE A:

1 B,
2 C VECTOR,
2 D SCALAR,
1 E,
2 H EVENT,
2 G INTEGER,
1 H BIT(16);

DECLARE A A-STRUCTURE;

MINOR_STRUCT = E;

M = G;

B_BIT = H;

structure template

unqualified declaration

references to parts of
structure A

5.2.2 Qualified Structures

If a structure is qualified, then any part of it, either minor structure or terminal, is referenced as follows. First, the name of the part of the structure is taken. Then the "branches" of the structure tree are traversed back from it to the "root" or major structure (see Section 4.3). On passing through each "fork" or minor structure, the name is prefixed with a period and then with the name of that minor structure. This process ends with the prefixing of the major structure name. If a minor structure is being referenced, the resulting "qualified" name becomes syntactically a <structure var name>. If a terminal is referenced, then syntactically it becomes a <\$var name>, where \$ stands for any of the types, arithmetic, bit, character, or event, as appropriate to the attributes of its definition in the template.

EXAMPLE:

STRUCTURE A:

```
1 B,  
2 C VECTOR,  
2 D SCALAR,  
1 E,  
2 H EVENT,  
2 G INTEGER,  
1 H BIT(16);
```

] structure template

```
DECLARE Z A-STRUCTURE;
```

] qualified declaration

```
MINOR_STRUCT = Z.E;
```

```
M = Z.G;
```

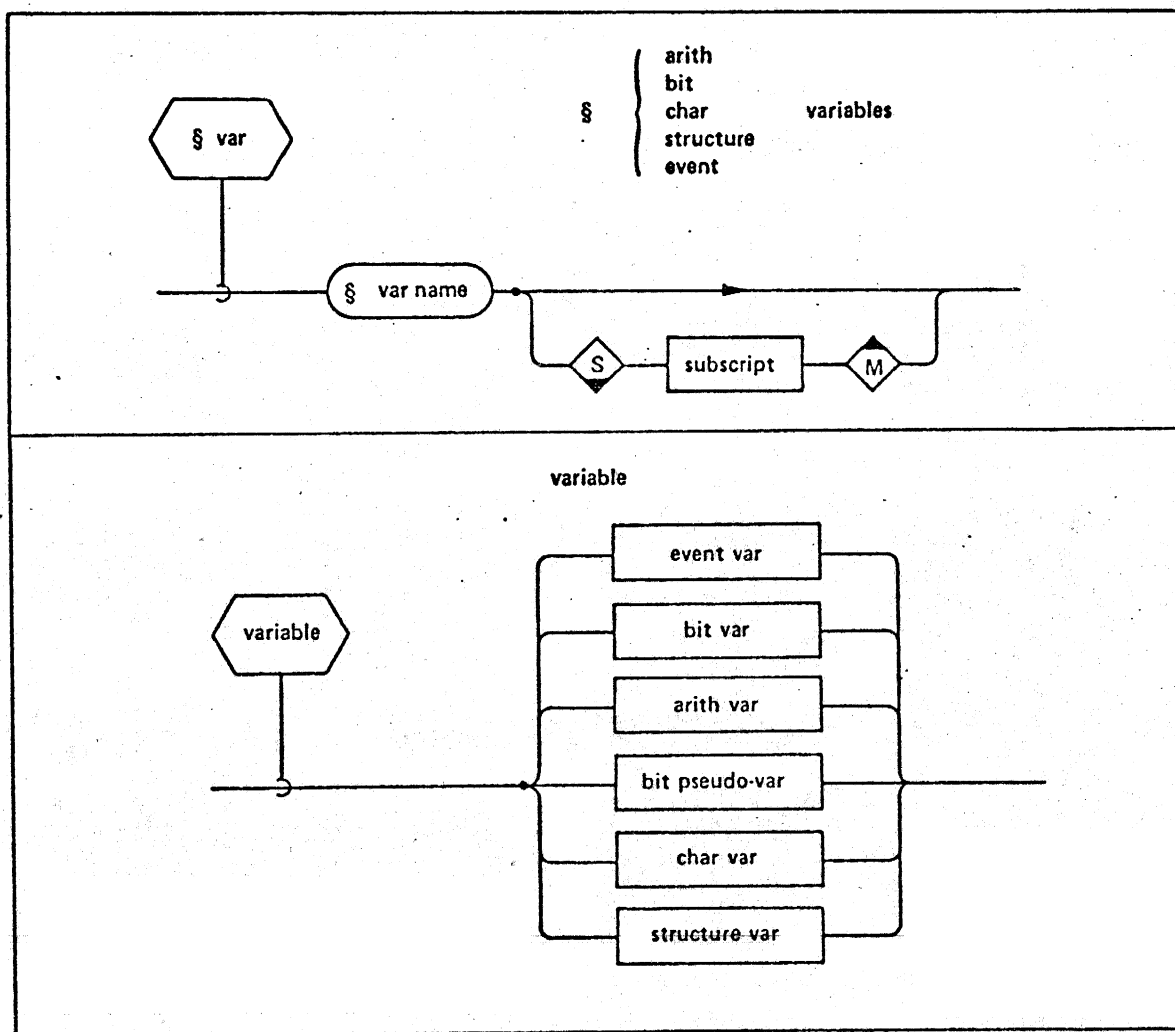
```
B_BIT = Z.H;
```

] references to parts of structure Z

5.3 Subscripting

For the remainder of this Section, unsubscripted variable names are denoted syntactically by $\langle \$var\ name \rangle$, where $\$$ stands for any of the types arithmetic, bit, character, event, or structure. It is convenient to introduce the syntactical terms $\langle \$var \rangle$ to denote a subscripted or unsubscripted $\langle \$var\ name \rangle$, and $\langle variable \rangle$ to mean any type of $\langle \$var \rangle$. $\langle bit\ pseudo-var \rangle$ is a reference to the SUBBIT pseudo-variable (see Section 6.5.4).

SYNTAX:

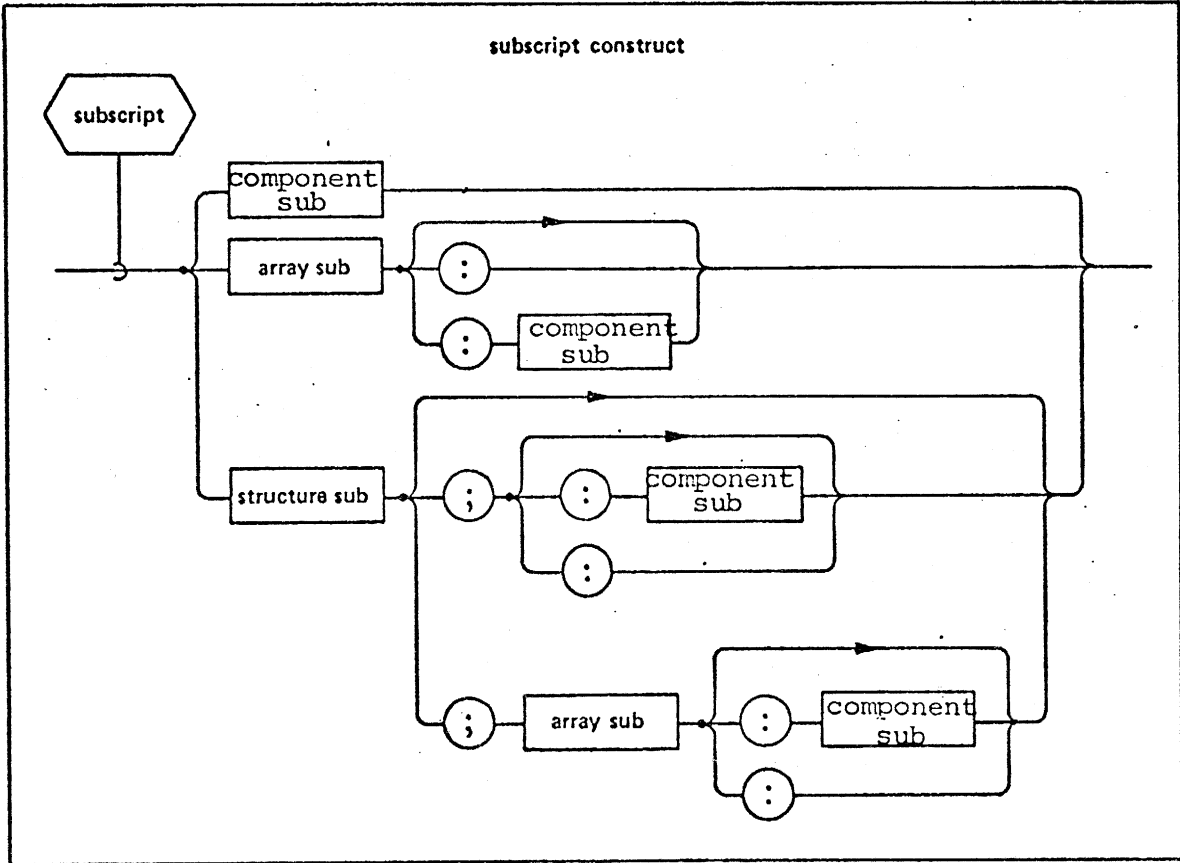


5.3.1 Kinds of Subscripting

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

- <component sub> can be applied to simple variables and structure terminals which have one or more component dimensions (i.e. made up of distinct components). The applicable types are vector, matrix, bit and character (e.g. C_8).
- <array sub> can be applied to any arithmetic, bit, character, and event variables which are given array specification in their declaration. This includes both simple variables and structure terminals (e.g. I_{16}).
- <structure sub> 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.

SYNTAX:



5.3.1.1 Subscripting Data Types and Arrays of Data Types.

Subscripting of an unarrayed vector, matrix, bit or character is accomplished by use of the form <component sub> and references a single component. Subscripting of an array of integers, scalars, or events is accomplished by use of the form <array sub> and references a single data element (e.g. $\{A\}_I$).

Subscripting of an array of vector, matrix, bit, or character type has three forms:

- a <component sub> will yield an array (of same array dimension) of the specified components. An array of matrices subscripted with a * for one index will yield an array of vectors; an array of scalar vectors will yield an array of scalars, etc. (e.g. $[\bar{V}]_I$ or $[M]_{4,3}$).
- the form <array sub>:* (where the * is optional) will yield all of the data elements of the specified array component (e.g. $\bar{B} = [\bar{V}]_{4,:}$);
- the form <array sub>:<component sub> will yield the specified element of the specified array. (e.g. $\bar{B} = [M]_{4:*,3}$; or $C = [M]_{4:2,3}$);

$\bar{M}_{4:,3}$;

$M_{4:2,3}$;

5.3.1.2 Subscripting Unarrayed Structure Terminals. The use of the form <structure sub> specifies which structure copy is referenced to find the given integer scalar or event type structure terminal (e.g. $A.B_{25}$).

If the structure terminal is of vector, matrix, bit or character type, then

- the form <component sub> will yield all of the copies (in each structure copy) of the specified component.
- the form <structure sub>; will yield the structure terminal of the specified structure.
- the form <structure sub>;<component sub> will yield the specified component of the structure terminal of the specified structure.

5.3.1.3 Subscripting Arrayed Structure Terminals. If the structure terminal is of integer, scalar, or event type, then:

- the form <array sub> references the specified array element of the terminal of each copy of the structure.
- the form <structure sub>; references the arrayed structure terminal in the given structure copy.
- the form <structure sub>; <array sub> references the specified array element of the terminal of the specified copy of the structure.

If the structure terminal is an array of vector, matrix, bit, or character type then:

- the form <component sub> references the specified component of each array element of the terminal of each structure copy.
- the form <array sub>; references the specified array component (i.e. vector, matrix, bit or character type) of the terminal of each copy of the structure.
- the form <structure sub>; references the arrayed matrix, vector, bit, character data type of the terminal of the specified copy of the structure.
- the form <array sub>;<component sub> references the specified component of the specified array of the terminal of each copy of the structure.
- the form <structure sub>;<component sub> references the specified component of each array element of the terminal of the specified copy of the structure.
- the form <structure sub>;<array sub>; references the specified array component of the terminal of the specified copy of the structure.
- the form <structure sub>;<array sub>;<component sub> references the specified component of the specified array of the terminal of the specified copy of the structure.

EXAMPLES:

1. $M_{3,4}$ references the matrix-component in the third row, fourth column.
2. $A_{2,3,4}$ references a scalar or integer array element in the second plane, third row, fourth column of array A.
3. $A_{2,3,4:3,4}$ references the component in the third row, fourth column of the matrix located in the second plane, third row, fourth column of the array, A.
4. $BIT_{16}(A)$ references the 16th bit in the bit representation of A.
5. $TEXT_8$ references the 8th character in the string.
6. $M_{3,4}^*$ references the matrix in the third row, fourth column of the array of matrices, [M].

7. STRUCTURE A:

```
    1 B,  
      2 C ARRAY(4,4) MATRIX(3,3),  
      2 D INTEGER,  
    1 E,  
      2 G VECTOR(3),  
    1 F BIT(1);  
DECLARE A A-STRUCTURE(50);
```

The following examples refer to the above structure template and declaration.

a. $C_{8; \overset{4,2}{\downarrow}:1,2}$

This represents the scalar component in the first row, second column of the matrix which occupies the 4,2 position in the array C. This array is in the 8th copy of A.

b. $\{G\}_2$

This represents the second component of the vector G in all copies of A.

c. \dot{F}_{25} ;

This represents the single 1-bit, bit-string in the 25th copy of A.

d. $\{\dot{C}\}_{23;4,*}$:

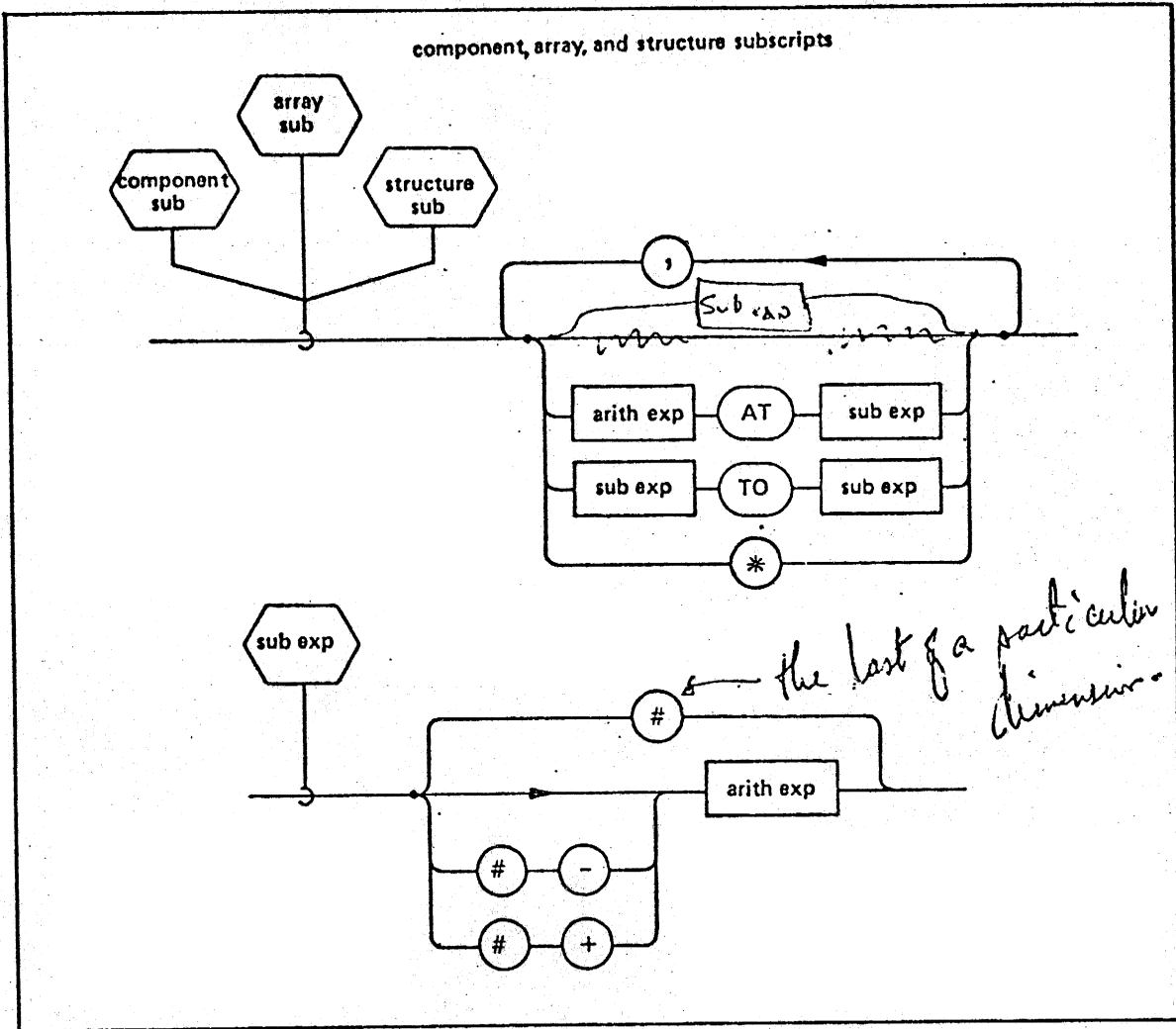
This represents the array of all of the matrices (specified by *, see Section 5.3.2) in the "4th row" of the array C, in the 23rd copy of A.

5.3.2 Forms of Subscripting

A <structure sub>, <array sub>, or <component sub> consists of a series of subscript expressions separated by commas. Each subscript corresponds to the particular structure, array or component dimension to which it is attached. The form <sub exp> specifies the index of one component, array element, or structure copy to be selected. The TO phrase may be used to reference (or partition) a set of elements by specifying the lower and upper index limits respectively. Similarly, the AT phrase may be used to reference a set of elements by specifying the size (or length) of the set, and the lower index limit respectively. The use of a number sign (#) results in the value of the upper limit of the particular index.

The use of the * indicates "all of a particular index" and can be used to establish a cross section of a matrix or an array.

SYNTAX:



A ARRAY (25)

$A_{\#}$ means A_{25}

EXAMPLES:

$\overset{*}{M}_4$ AT 5, 4 AT 7

$\bar{M}_{*,4}$

$[\bar{V}]_{2,*}$:

$\overset{\cdot}{B}_5$ TO 10..

$[\overset{\cdot}{A}]_{\text{two } (P+2), 1 \text{ TO } 3:4 \text{ TO } \#}$

BATP

5.3.3 The Arrayness of Variables and Expressions

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

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

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

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

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

The arrayness of HAL/S expressions originates from that of their operands, and thus from the `<$var>`s appearing in them. Although the forms of subscript distinguish between array dimensions and structure copies, no distinction is made between them as far as arrayness matching is concerned.

```
EXAMPLE:      STRUCTURE Z:
                1 B ARRAY(5),
                1 C SCALAR;
DECLARE A Z_STRUCTURE(10),
                C ARRAY(10,5);

[C] = [A.B] + [C];      /* ARRAYNESS IS 10,5 */
```

y = 2
 Success

5.4 The Natural Sequence of Elements

There are several kinds of operations in the HAL/S language which require <\$var>s with multiple components, array elements, and structure copies, and also <expression>s, to be unraveled into a linear array or string of data values. The reverse process of "reraveling" a linear array or string also occurs. The two major occurrences are in I/O (see Section 10) and conversion functions (see Section 6.5). The order of unraveling is called the "natural sequence".

5.4.1 The Natural Sequence of Major and Minor Structures

- Each copy of the major or minor structure is unraveled in turn, in order of increasing index. (e.g. A.B₁; A.B₂; A.B₃; etc.)
- Each structure terminal defined under the major or minor structure is unraveled in turn, in order of their appearance in the structure template.
- Each structure terminal is unraveled according to the rules given below.

5.4.2 The Natural Sequence of Simple Variables and Structure Terminals

- If a structure terminal has multiple copies, each copy is unraveled in turn, in order of increasing index.
- If the simple variable is arrayed, each array dimension is unraveled in turn, starting from the leftmost defined dimension, and in order of increasing index. (e.g. ~~[B]_{1,1}~~ ~~[B]_{1,2}~~ ~~[B]_{1,3}~~ ~~[B]_{2,2}~~
~~[B]_{2,3}~~ } $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,2}$
 $B_{2,3}$)
- Integers, scalars, characters, bits and events are considered as having only one component data value.
- Vectors are unraveled component by component, in order of increasing index.
- Matrices are unraveled row by row, in order of increasing index. The components of each row are unraveled in turn, in order of increasing index. This process is similar to that for arrays above.

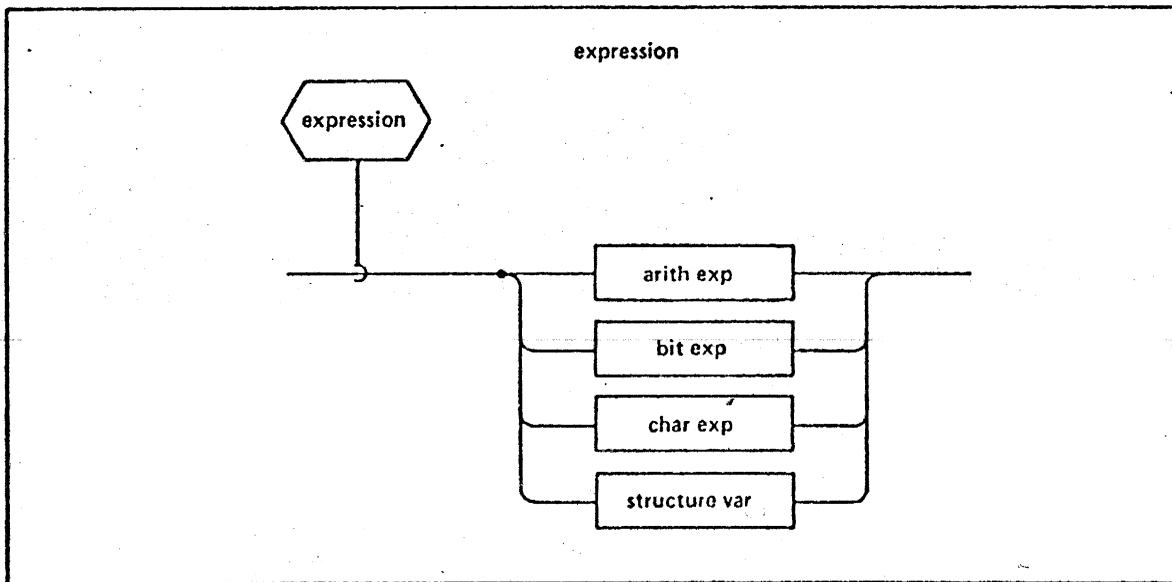
6.0 DATA MANIPULATION AND EXPRESSIONS

An expression is an algorithm used for computing a value. In HAL/S, expressions are formed by combining 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. Expressions are divided into three major classes according to their usage: regular expressions, conditional expressions, and event expressions.

6.1 Regular Expressions

Regular expressions comprise arithmetic expressions, bit expressions, and character expressions, together with structure variables. An <expression> can appear in an assignment statement, as an input argument of a procedure or function block, or in a WRITE statement.

SYNTAX:



6.1.1 Arithmetic Expressions

An <arith exp> is a sequence of <arith operand>s (see Section 6.1.4.1) separated by arithmetic operators, and possibly preceded by a unary plus or minus.

The following table summarizes the precedence (i.e. order of operation) rules for arithmetic operators:

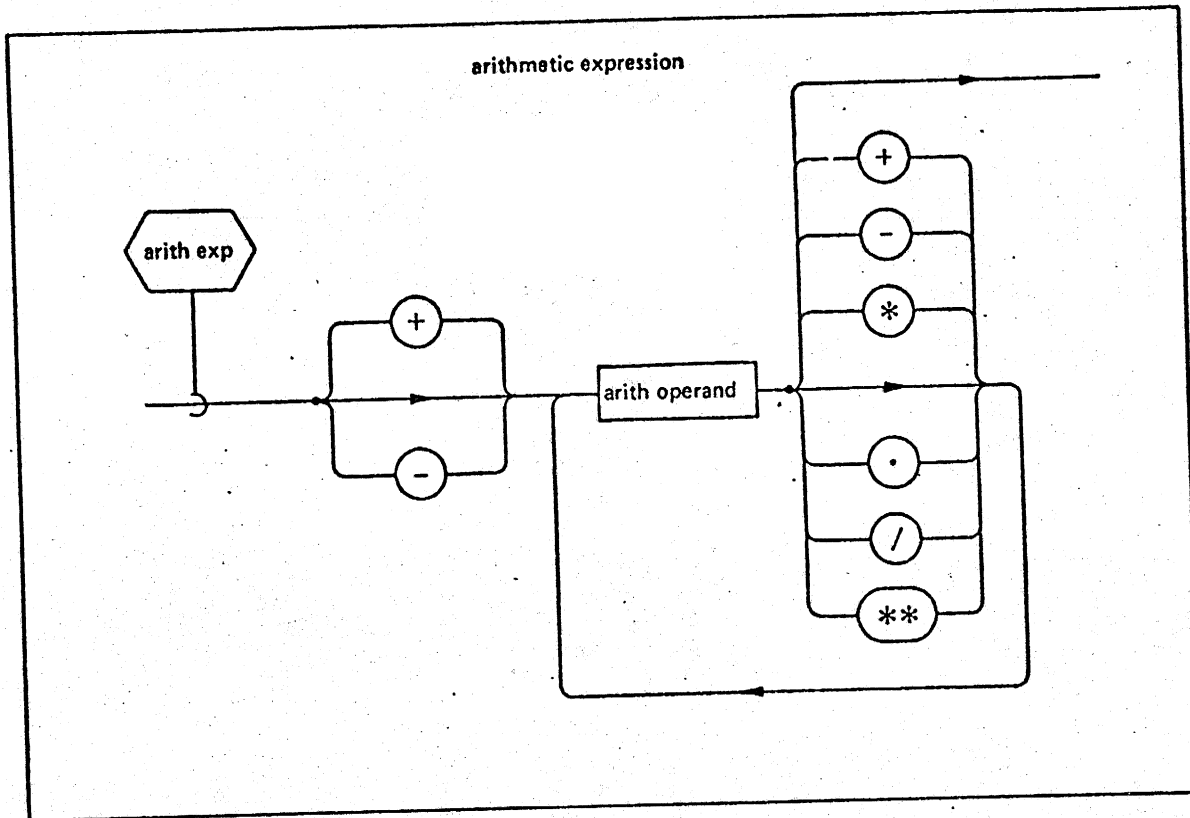
OPERATOR	PRECEDENCE
**	1 (FIRST)
<>	2
*	3
.	4
/	5
+, -	6 (LAST)

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

- operators **, / are evaluated right-to-left;
- operators <> are evaluated so as to minimize the total number of elemental multiplications required;
- all other operators are evaluated left-to-right.

Table C. of the appendix summarizes the results of a given operator applied to all possible types of <arith operand>s.

SYNTAX:



EXAMPLES:

$$I + J - (K+2)^3$$

INTEGER EXPRESSION: I, J, K INTEGERS

$$\bar{M} \cdot (\bar{M} * \bar{N})$$

VECTOR EXPRESSION

$$R^P$$

SCALAR EXPRESSION: R, P SCALARS

$$(\bar{M} + \bar{N})^{-2}$$

MATRIX EXPRESSION

$$A/B \cdot C$$

MULTIPLY DONE BEFORE DIVIDE

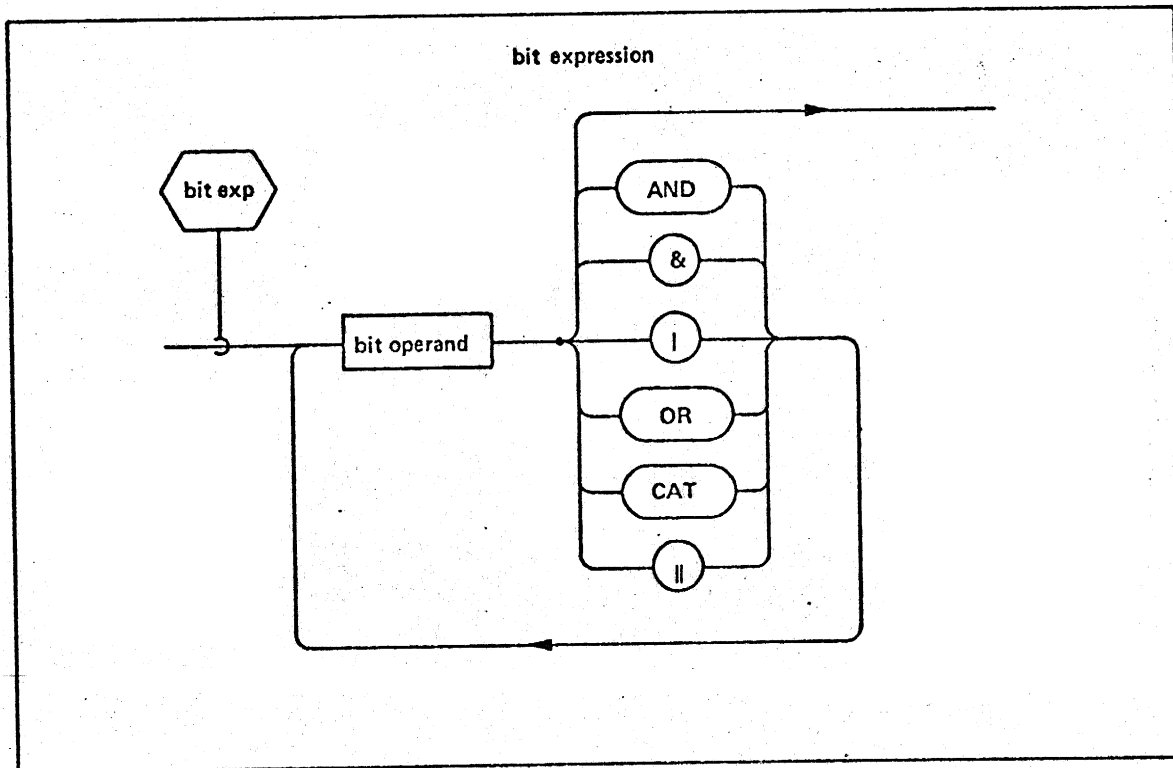
6.1.2 Bit Expressions

A <bit exp> is a sequence of <bit operand>s (see Section 6.1.4.2) separated by bit operators whose order of evaluation is:

OPERATION	OPERATOR	PRECEDENCE
Catenation	CAT,	1 (FIRST)
Logical Intersection	AND, &	2
Logical Union	OR,	3 (LAST)

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

SYNTAX:



EXAMPLES:

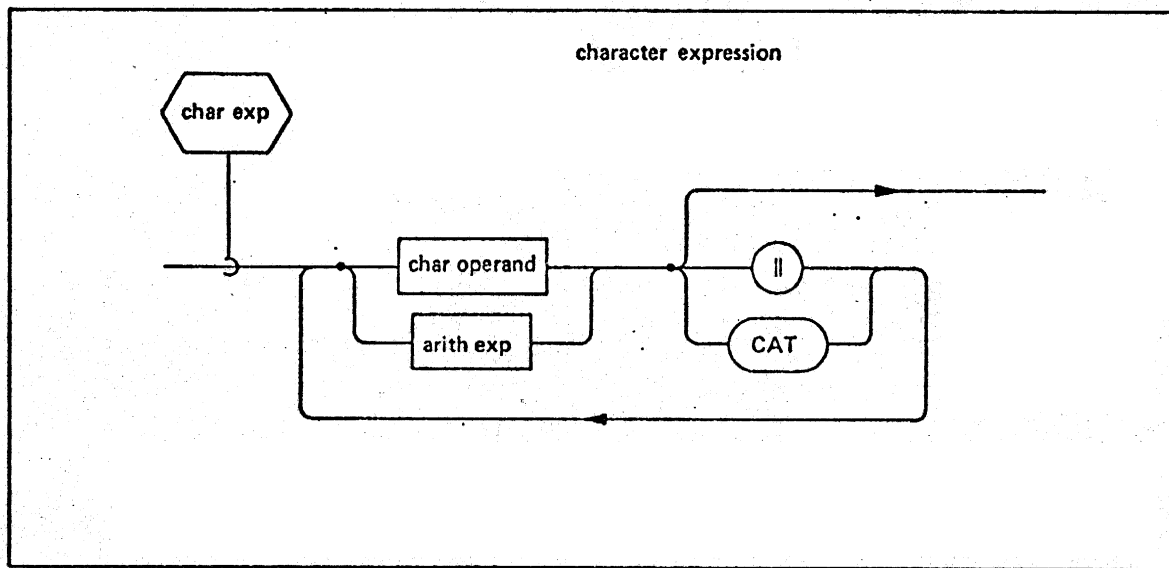
$\dot{B} \ \& \ \dot{C} \ || \ \dot{D}$

$\dot{A} \ OR \ (\dot{B} \ AND \ \dot{C})$

6.1.3 Character Expressions

A <char exp> is a sequence of operands separated by the operators: CAT or ||. Each operand may be a <char operand> (see Section 6.1.4.3) or an integer or scalar <arith exp>. The sequence of catenations is evaluated from left-to-right.

SYNTAX:



EXAMPLES: Q || I || QQ;

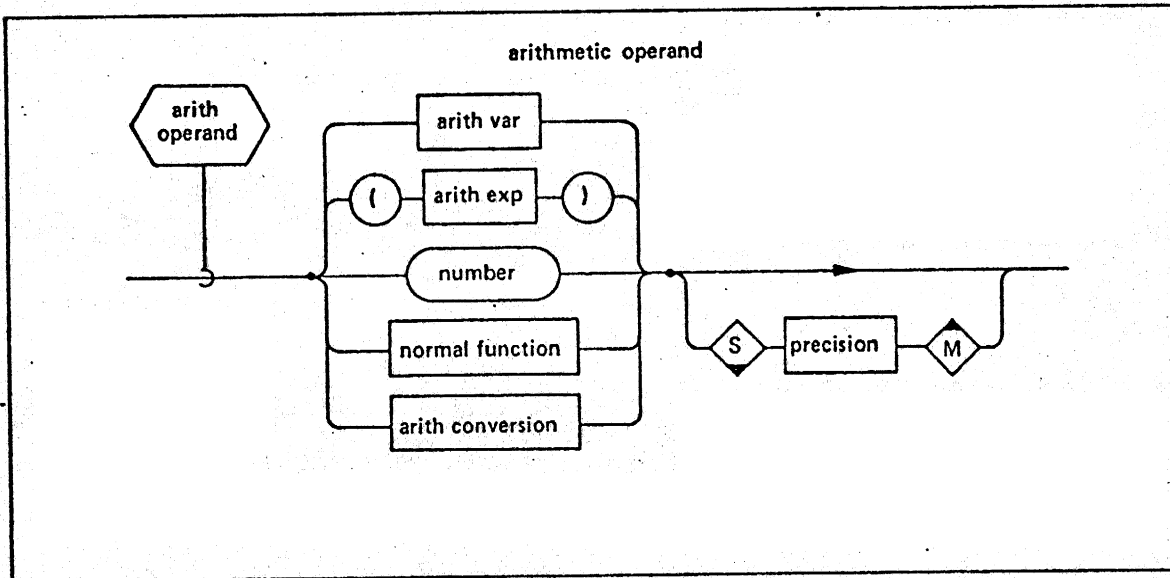
 TEXT || 'HELP' || (A/S) || (B || C)

6.1.4 Regular Expression Operands

Operands of the appropriate type are used with operators to form regular arithmetic, bit or character expressions. These operands include <arith operand>s, <bit operand>s, and <char operand>s.

6.1.4.1 Arithmetic Operands. An <arith operand> may be an arithmetic variable, an arithmetic expression enclosed in parenthesis, a <normal function> of the appropriate type, an <arith conversion> function, or a literal <number>. Precision may be specified by a <precision> subscript (see Section 6.7).

SYNTAX:



EXAMPLE: (A+B)@DOUBLE

SIN(X)

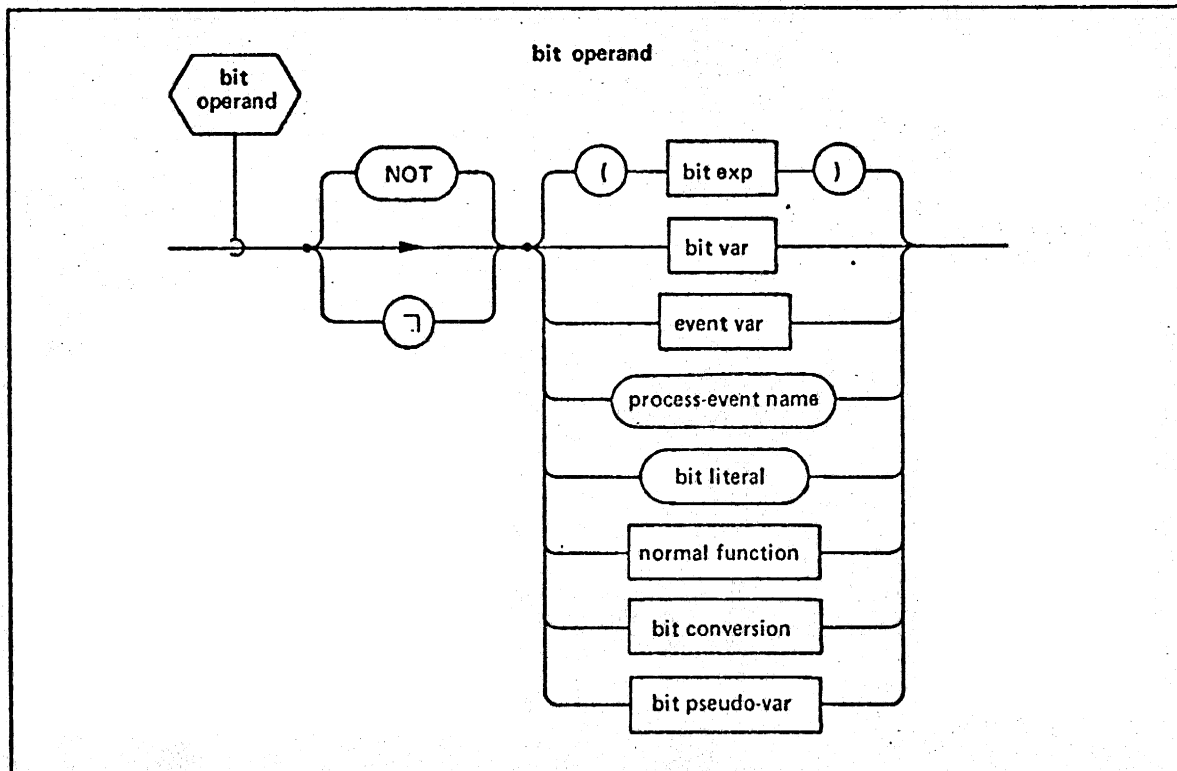
[A]_{1 TO 5}

INTEGER(X²)

36.047

6.1.4.2 Bit Operands. A <bit operand> may be a <bit var>, a <bit exp> enclosed in parenthesis, a <bit literal>, a <normal function> of bit type, a <bit conversion> function, or a <bit pseudo-var>. In real time programming, a <bit operand> may be an <event var> or a <process-event name> (see Section 8.9). Any form of <bit operand> may be prefaced by NOT or $\bar{\quad}$, causing its logical complement.

SYNTAX:



EXAMPLES:

\bar{B}
BIT'^N11010110'

\bar{B}

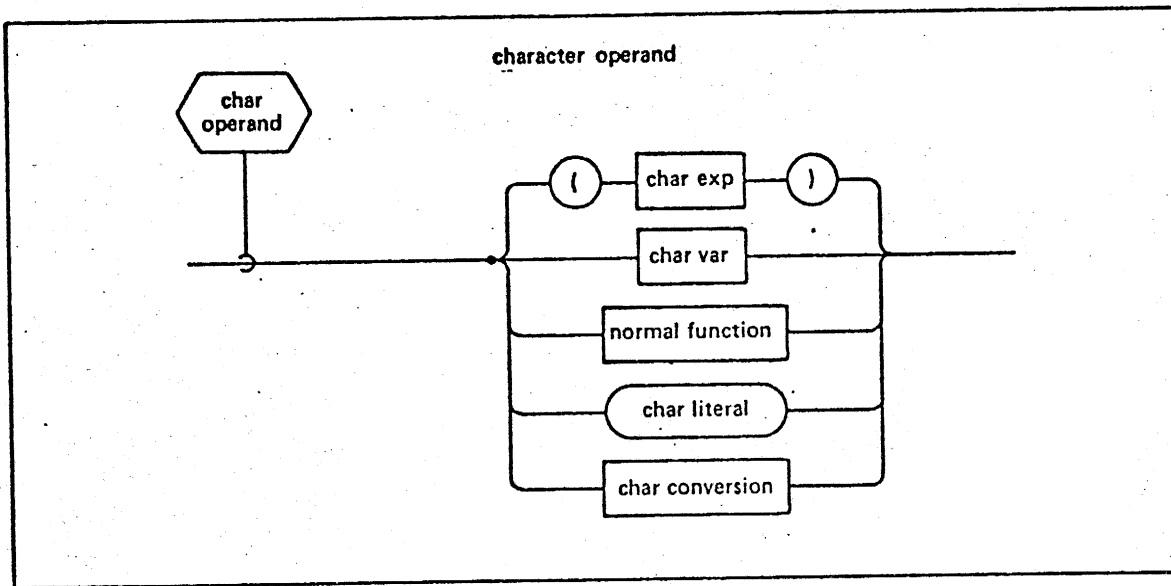
BIT(A)

(A|C)

\bar{C}_1 TO 8

6.1.4.3 Character Operands. A <char operand> may be a <char var>, a <char exp> enclosed in parenthesis, a <char literal>, a <normal function> of character type, or a <char conversion> function.

SYNTAX:



EXAMPLE:

'DELTA'

(STATUS || 'O.K.')

CHARACTER(I+J)

6.1.5 Array Properties of Expressions

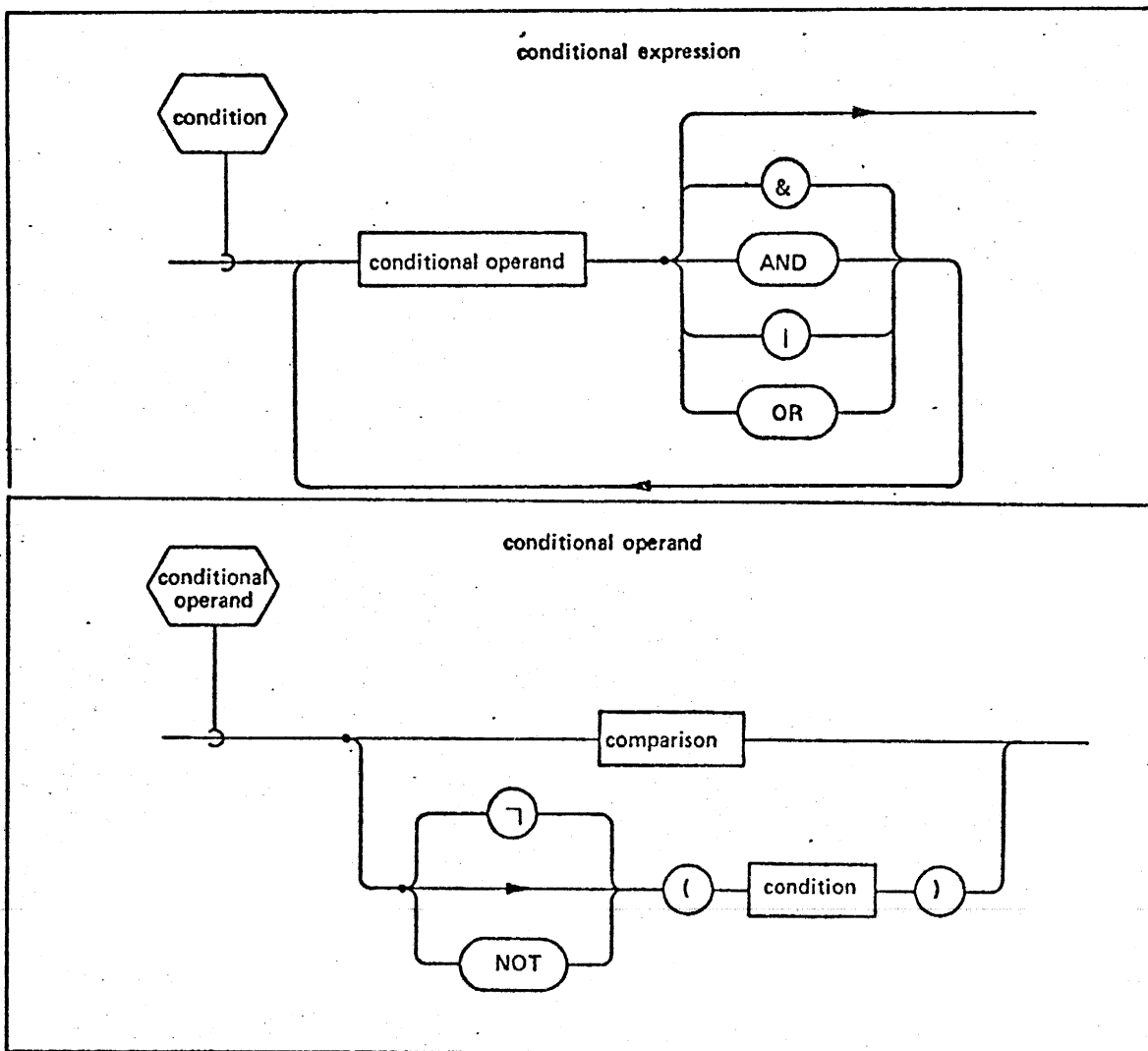
Any regular expression may have an array property by virtue of possessing one or more arrayed operands. The evaluation of an arrayed regular expression implies an element-by-element evaluation of the expression. If only one operand is arrayed, then evaluation of the operation using the unarrayed operand and each element of the arrayed operand is implied. If more than one operand is an array of equal dimension, evaluation of the operation for each of the corresponding elements is implied. In all cases, the result is an array of the same dimension as the operand array.

6.2 Conditional Expressions

A <condition> is a sequence of <conditional operand>s separated by logical operators, whose order of evaluation is:

OPERATION	OPERATOR	PRECEDENCE
Logical Intersection	AND, &	1 (FIRST)
Logical Union	OR,	2 (LAST)

SYNTAX:



EXAMPLES:

$\neg (A > B) \mid (A > C)$

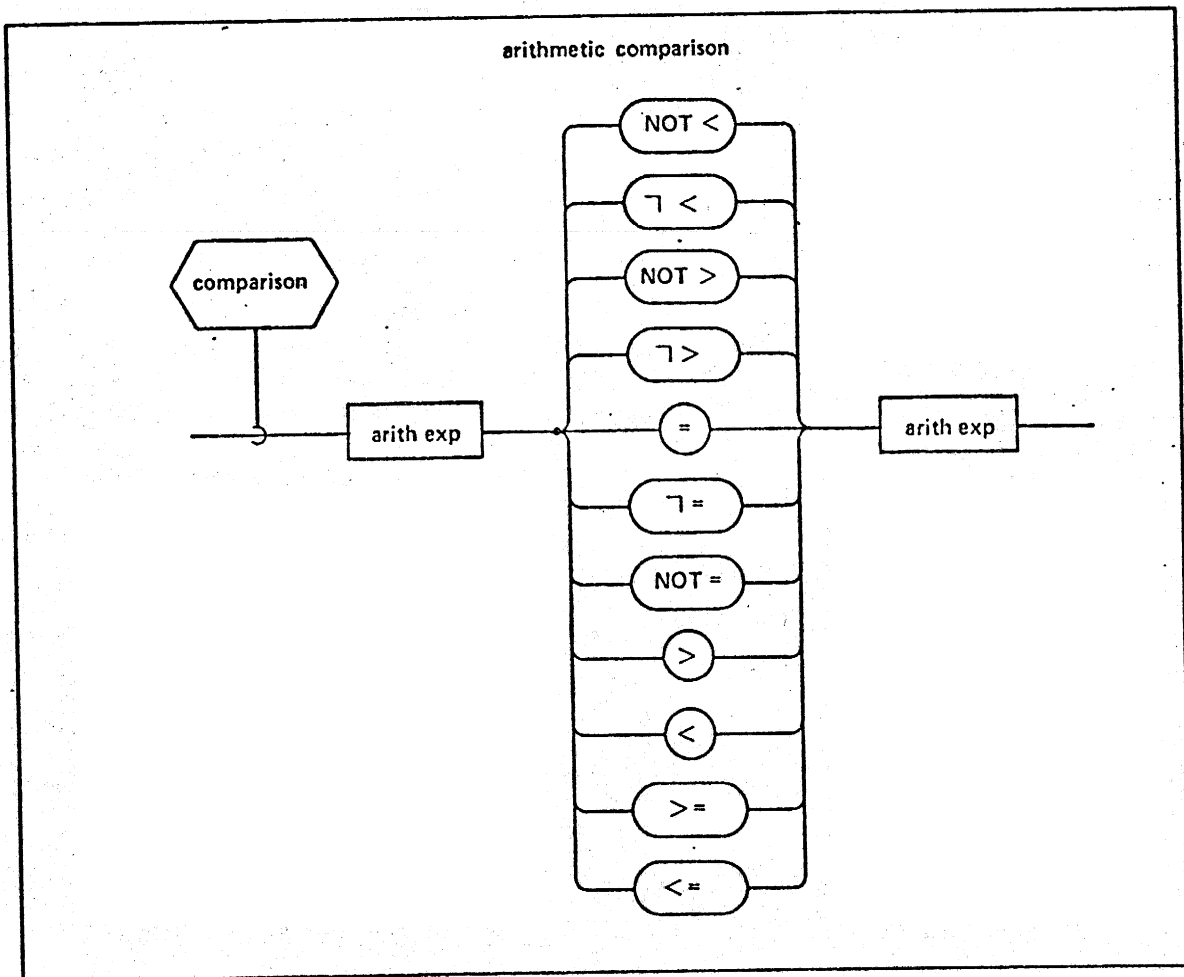
$(A \leq B)$

$X > 100 \text{ AND } \neg (Y < 3 \text{ OR } Z > 2)$

6.2.1 Arithmetic Comparisons

An arithmetic <comparison> is a comparison between two <arith exp>s whose types must match (except for mixed integer and scalar operands when the integer operand is converted to scalar). Valid combinations of types of <arith exp>s for comparison may be found in Appendix C. If the operands are vectors or matrices, the operator must be =, \neq , NOT=, and is compared element-by-element.

SYNTAX:



EXAMPLES:

$I > J$

$(M+N) \text{ NOT } < 36$

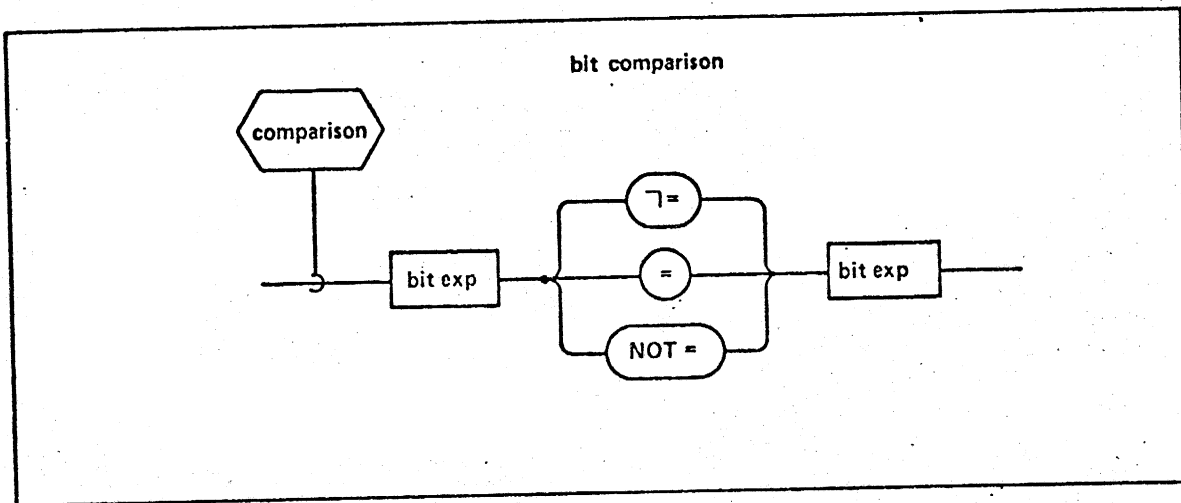
$\overset{*}{K} \neq \overset{*}{L}$

$I \leq (A + \bar{P} \cdot \bar{V})$

6.2.2 Bit Comparisons

A bit comparison is a comparison between two <bit exp>s which are said to be equal if they have identical bit patterns. If the operands have different lengths, the shorter operand is left padded with binary zeros to match the length of the longer <bit exp>.

SYNTAX:



EXAMPLES:

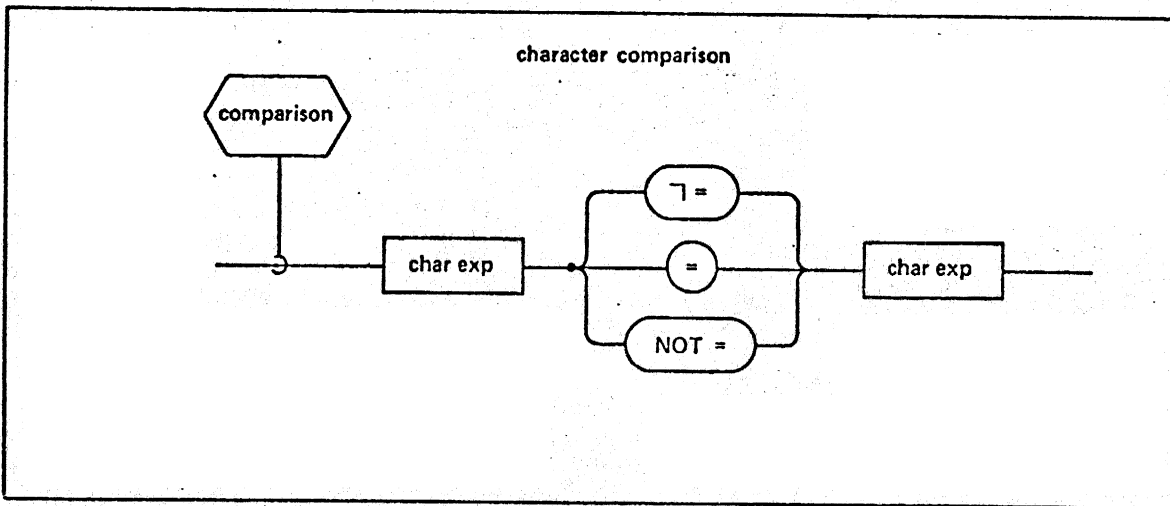
$\dot{B} \neq \text{BIN}'110'$

$\dot{D} = \dot{E}$

6.2.3 Character Comparisons

A character comparison is a comparison between two <char exp>s. If the operands have different lengths, the <char exp> of shorter length is right padded with blanks to match the length of the longer operand.

SYNTAX:



EXAMPLES:

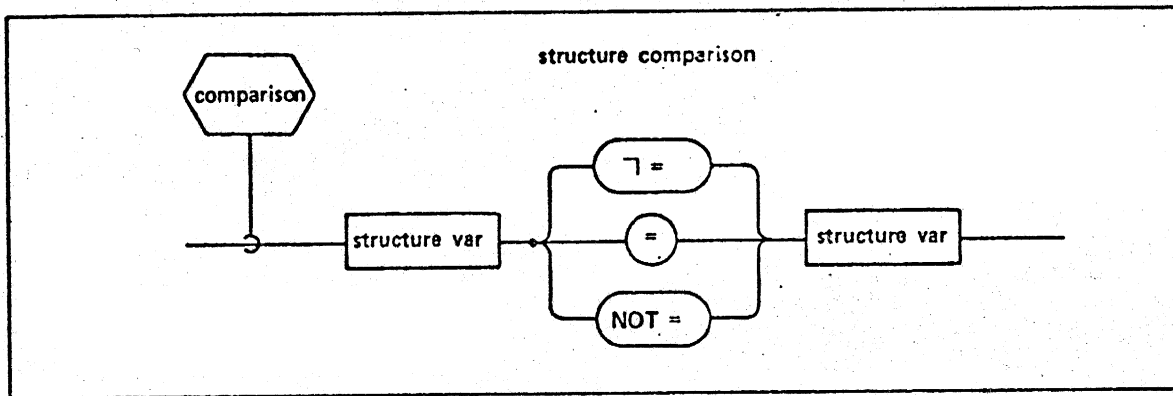
'C' = 'A'

'S' ≠ 'STOP'

6.2.4 Structure Comparisons

A structure comparison is a comparison between two <structure var>s whose tree organizations are identical in all respects and whose number of copies are equal. If the <comparison> operator is =, the result is TRUE only if it is TRUE for each copy; if the <comparison> operator is \neg = or NOT=, the result is TRUE if it is TRUE for at least one copy.

SYNTAX:



6.2.5 Comparisons Between Arrayed Operands

A <comparison> of any one of the forms described may have arrayed operands, although the <comparison> operators are restricted to =, \neg = and NOT=. The <comparison> is done on an element-by-element basis producing an unarrayed result. If the operator is = then the result is TRUE only if it is TRUE for all elements of the <comparison>; if the operator is \neg = or NOT= then the result is TRUE if it is TRUE for at least one element of the <comparison>.

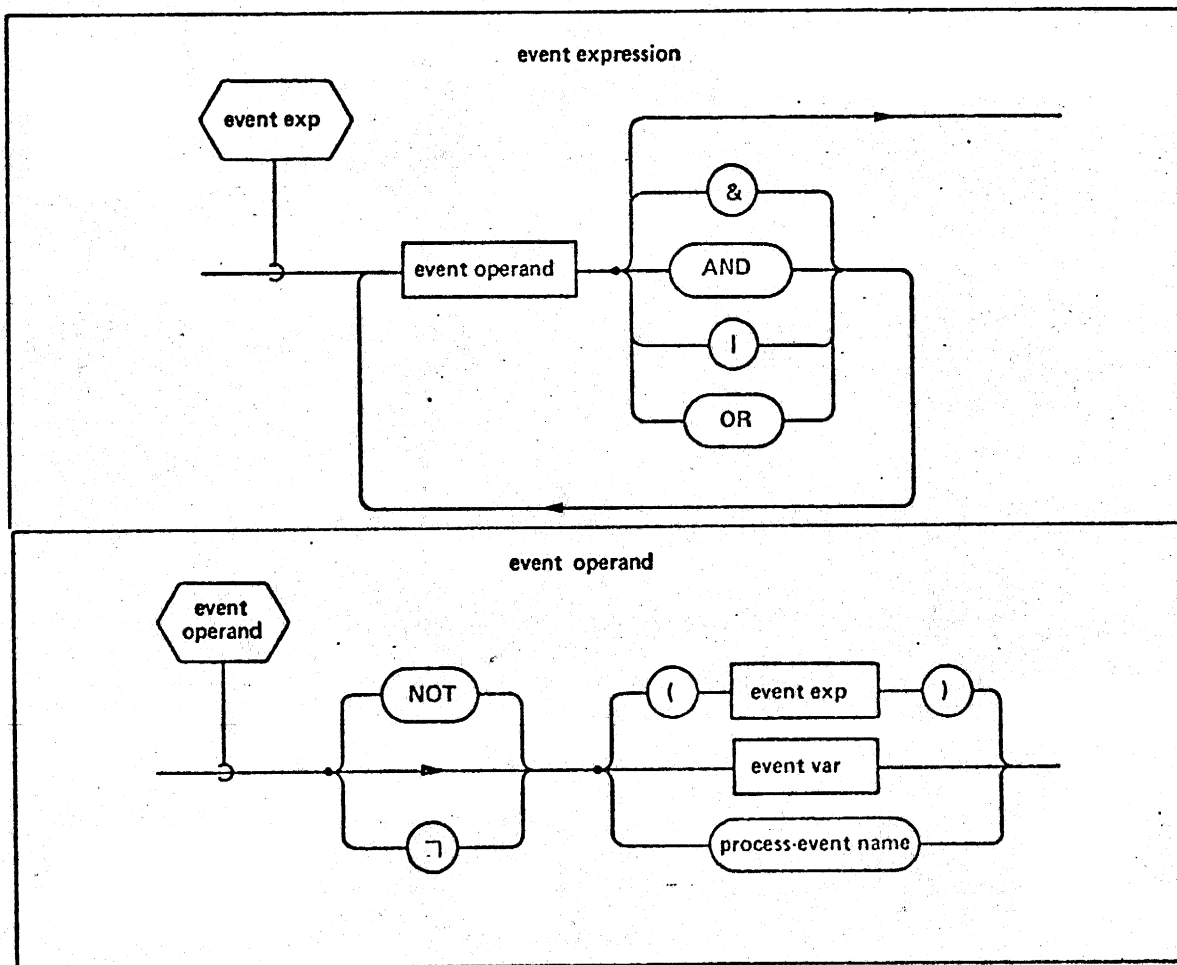
6.3 Event Expressions

An event expression, used in real time programming (see Section 8.), is an unarrayed sequence of <event operand>s separated by a subset of bit operators. The order of evaluation of each operation is dictated by operator precedence:

OPERATION	OPERATOR	PRECEDENCE
Logical Intersection	AND, &	1 (FIRST)
Logical Union	OR,	2 (LAST)

If two successive operations have equal precedence, they are evaluated from left-to-right. The <event operand> may be optionally prefaced by the logical complementing operators NOT or $\bar{\quad}$.

SYNTAX:



EXAMPLES:

ALPHA OR BETA

$\bar{(A \& B)}$

6.4 Normal Functions

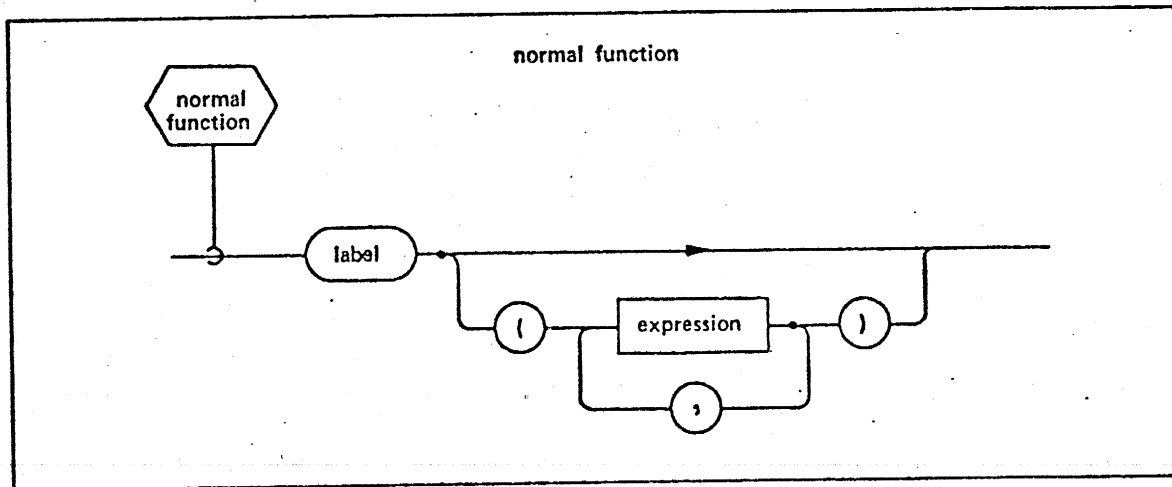
Section 6.1.1 through 6.1.3 have made reference to normal functions which are invoked by appearing as an operand in an expression. Normal functions fall into two classes:

- "built-in" functions named by <label> and defined as part of the HAL/S language (see Appendix B for a list of these functions);
- "user-defined" functions named by <label> and defined by the presence of <function block>s in <compilation>s.

Each <expression> or "input argument" of a normal function must match the corresponding input parameter of the function definition in type, terminal size, structure tree organization, etc.

If a user-defined function is invoked before it is defined by its <function block>, the name and type of the function must be declared at the beginning of the containing name scope.

SYNTAX:



EXAMPLES:

SIN(2X)
UNIT(\bar{V})
USER_COS(A)

6.5 Explicit Type Conversions

HAL/S contains a comprehensive set of function-like explicit conversions (see Appendix D.) some of which, called shaping functions, also have the property of being able to shape lists of arguments into arrays of arbitrary dimensions. HAL/S contains conversion functions to integer, scalar, vector, matrix, bit, and character types.

6.5.1 Arithmetic Conversion Function

The keyword INTEGER, SCALAR, VECTOR, or MATRIX gives the result type of the conversion. A <precision> specifier gives the precision of the result while a <subscript> specifier gives its dimensions. Any <expression> may be preceded by the phrase <arith exp># which denotes the number of times the <expression> is to be used in generating the result of the conversion.

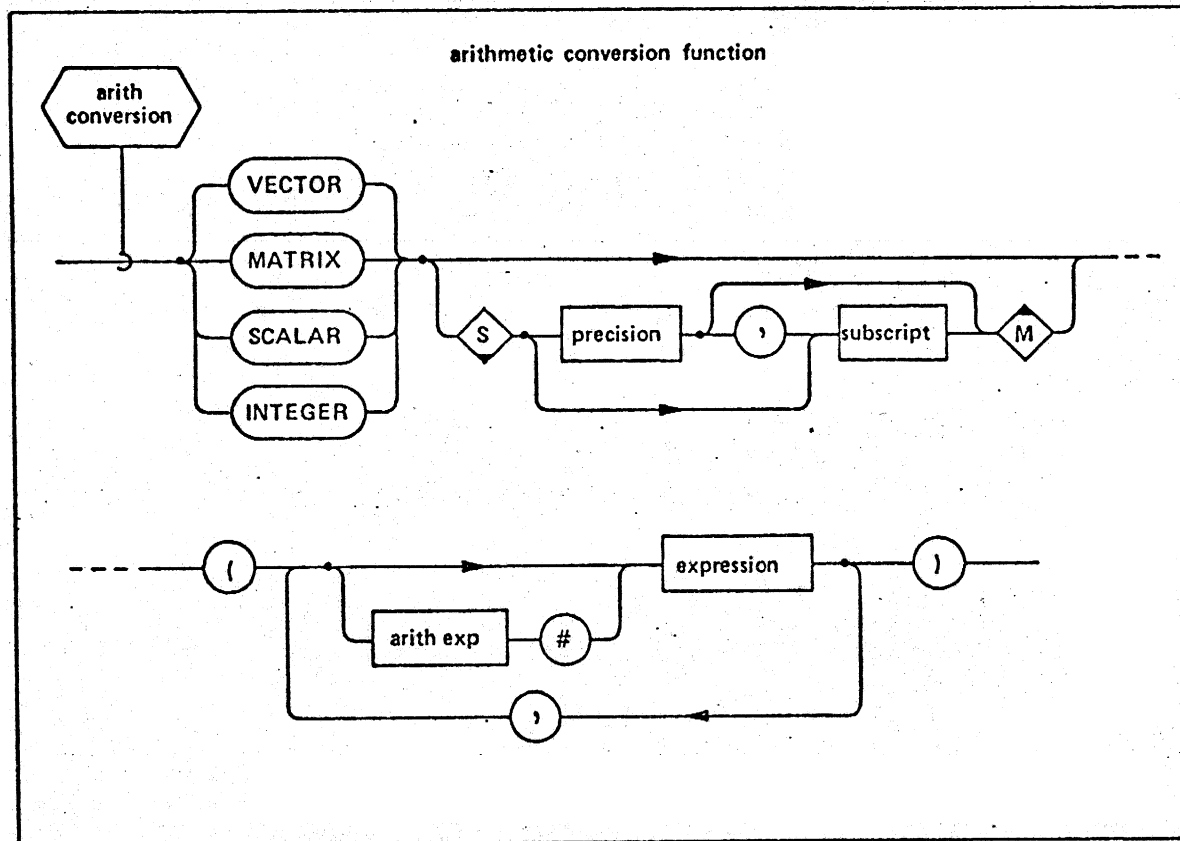
If INTEGER or SCALAR are subscripted, the <subscript>s denote the size of each array dimension produced. If there is no subscript, and if there is only one unrepeated arrayed argument, a linear (1-dimensional) array is produced. In all cases, INTEGER and SCALAR may have arguments of any type except structure.

A VECTOR <subscript> is an <arith exp> specifying the length of the resultant vector. If no subscript is specified, VECTOR produces a 3-vector result.

A MATRIX subscript has the form: <arith exp>, <arith exp> denoting the row and column dimensions respectively of the matrix result. If no subscript is specified, MATRIX produces a 3-by-3 matrix result.

VECTOR and MATRIX may have arguments of scalar, vector, and matrix type only.

SYNTAX:



EXAMPLES:

INTEGER_{2,2} (4*I+J)

SCALAR(A,B,C,15#D)

VECTOR_{2,4} (A,O,B,E)

MATRIX_{4,2} ([A])

VECTOR(X,Y,Z)

6.3.2 The Bit Conversion Function

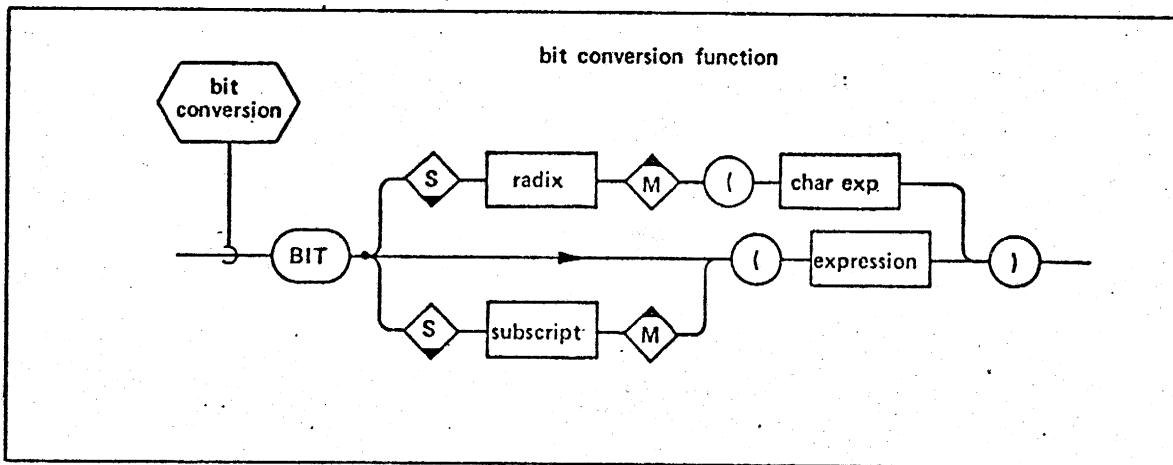
BIT converts an argument of integer, scalar, bit, or character type argument to a bit result. If the argument is arrayed, the conversion result is identically arrayed. <subscript> represents terminal subscripting upon the results of the conversion.

<radix> has the following possible forms:

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

The <char exp> consists of the legal digits listed to the right of each radix form above. The conversion generates binary representation of <char exp>.

SYNTAX:



EXAMPLES: BIT(I+J)
 BIT ~~1~~ 1 TO 8 (A)
 BIT@OCT ('657')
 BIT@HEX ('F2D')

6.5.3 The Character Conversion Function

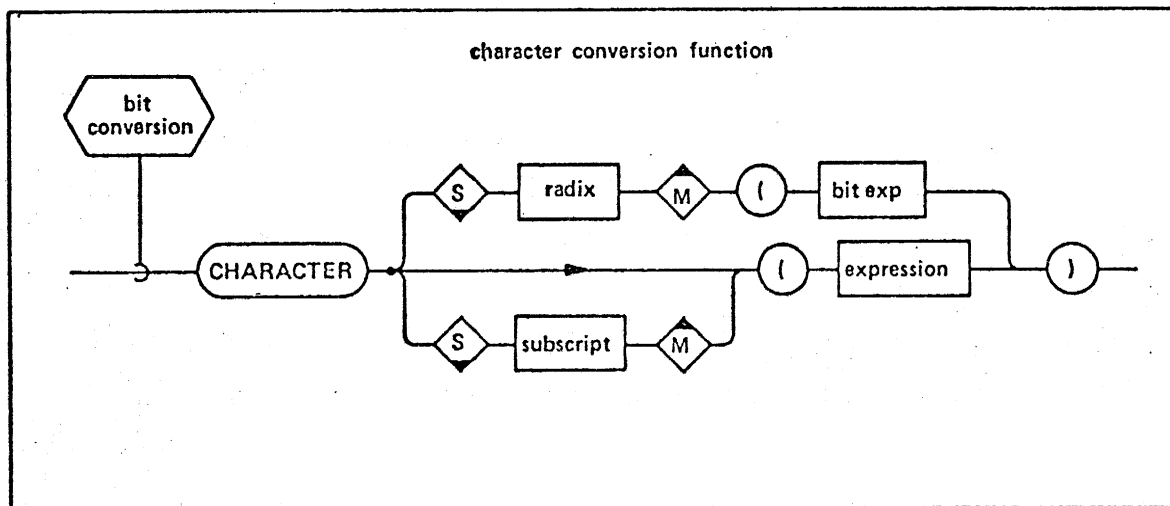
CHARACTER converts an integer, scalar, bit or character type argument to a character result. If the argument is arrayed, the conversion result is identically arrayed. <subscript> represents terminal subscripting upon the results of the conversion.

<radix> has the following possible forms:

@HEX	(hexadecimal string result)
@DEC	(decimal string result)
@OCT	(octal string result)
@BIN	(binary string result)

The value of <bit exp> is converted to the character string representation indicated above after left padding the value with binary zeroes as required.

SYNTAX:



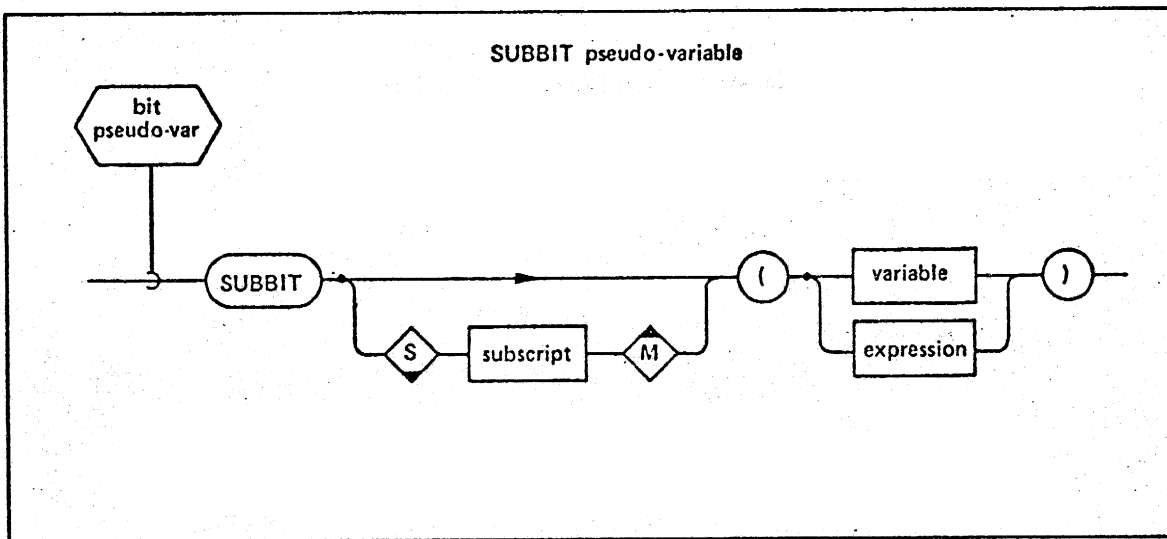
EXAMPLES:

CHARACTER_{@HEX} (B)
 CHARACTER (A_SCALAR)
 CHARACTER_{@DEC} (4567)

6.5.4 The SUBBIT Pseudo-Variable

The SUBBIT pseudo-variable allows access to other data types without conversion. It may appear in an assignment context with a <variable> argument, or as part of an <expression> as an operand of a <bit exp>. <subscript> represents terminal subscripting of the pseudo-variable.

SYNTAX:



EXAMPLE:

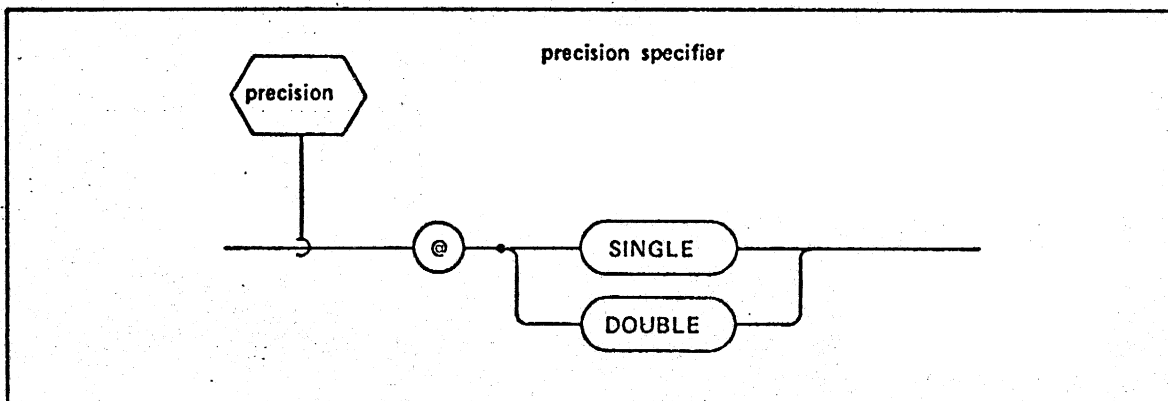
$\text{SUBBIT}_{5 \text{ TO } 8} (\dot{I}) = \dot{H} || \dot{A};$

$\dot{C} = \text{SUBBIT}_{1 \text{ TO } 8} (\dot{A});$

6.6 Explicit Precision Conversion

If <precision> is a subscript of an <arith operand>, a conversion to the precision specified takes place. If <precision> is a subscript of an <arith conversion> then the conversion result has the indicated precision. In referring to integer type, SINGLE implies a halfword and DOUBLE implies a fullword.

SYNTAX:



EXAMPLES:

$$A = E_{@SINGLE} + ((B+C_{@DOUBLE}) D)_{@SINGLE}$$

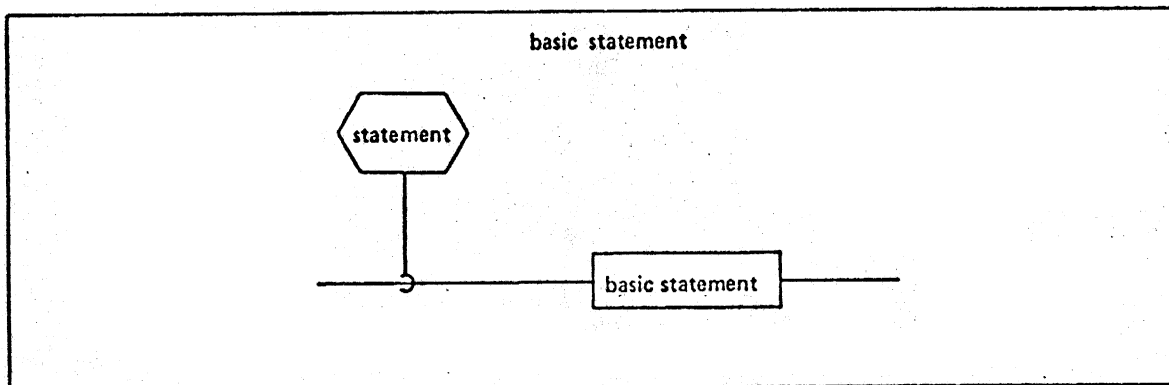
7.0 EXECUTABLE STATEMENTS

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

7.1 Basic Statement Definition

All forms of <statement> except the IF statement fall into the category of a <basic statement>. Not all of the <basic statement>s are described in this Section. Real time programming statements are described in Section 8., error recovery in Section 9., and input/output in Section 10.

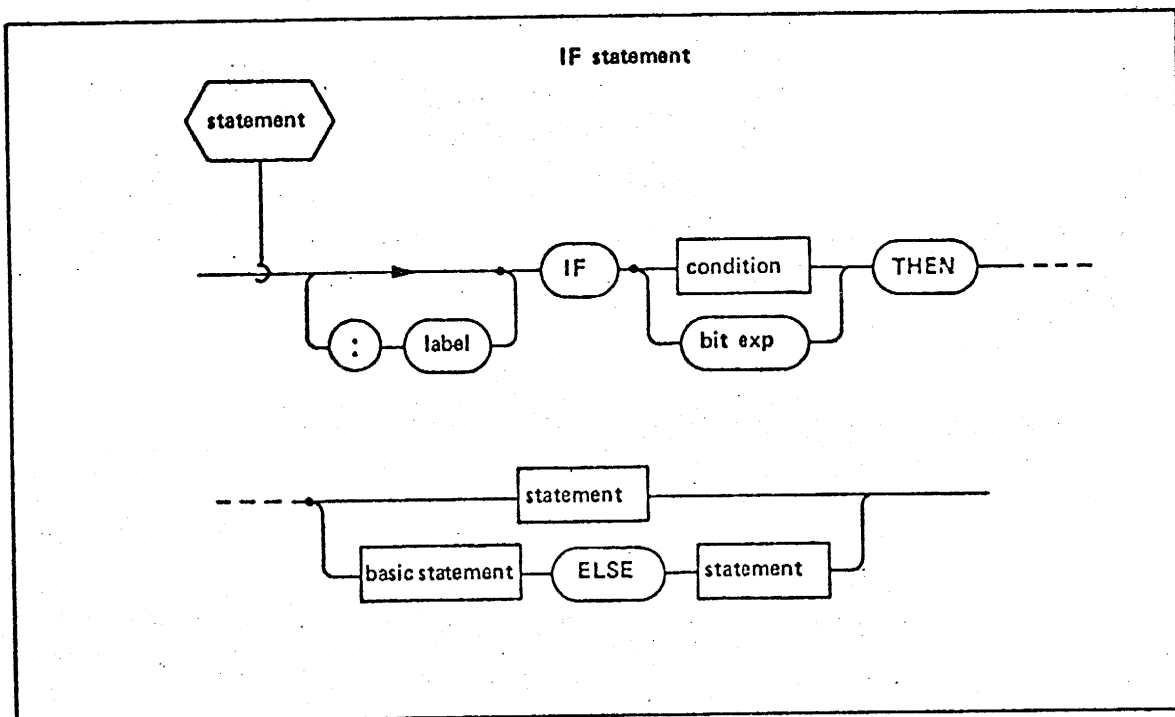
SYNTAX:



7.2 The IF Statement

The IF statement provides for the conditional execution of segments of HAL/S code. If the ELSE clause is present, then a second nested IF statement cannot appear preceding the key-word ELSE.

SYNTAX:



EXAMPLES: IF J>0 THEN K=1;
 ELSE K=2;

ABLE: IF K>=J THEN K=J-1;
 ELSE CALL TIME(\bar{V} ,T) ASSIGN(\bar{W});

IF A=B AND $M=N$ THEN DO;
 P=Q+1 ;
 D=E² ;
 END;

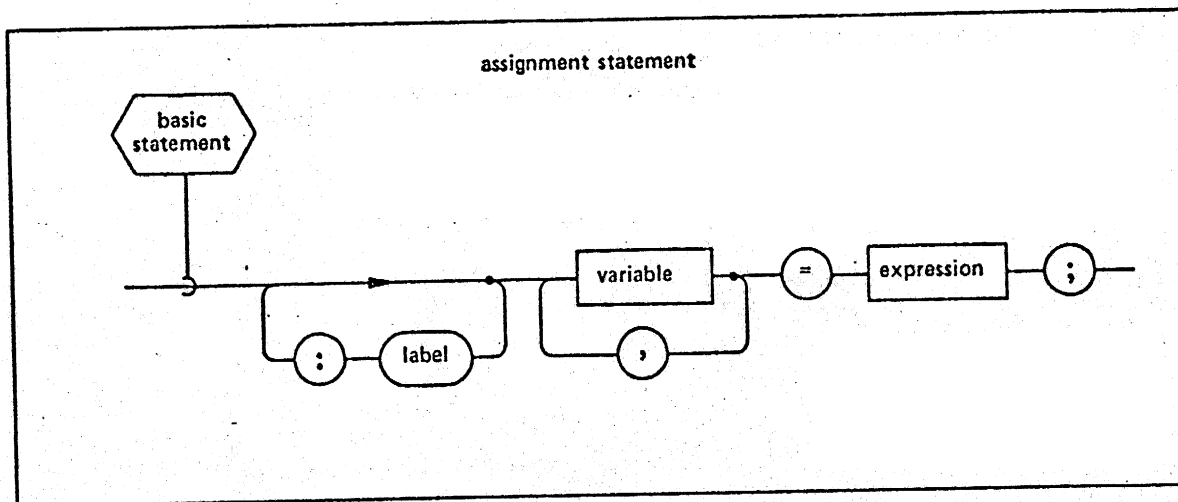
7.3. The Assignment Statement

The assignment statement is used to change the current value of a variable or a list of variables to that of an expression evaluated in the statement. In general, the dimensionality of <expression>s and <variable>s must match.

Execution is as follows:

- subscript expressions of the left-hand side are evaluated
- the <expression> is evaluated
- the values of the <variable>s on the left hand side are changed

SYNTAX:



EXAMPLES: ETA, KAPPA=LAMBDA+1;

SUM_ARRAY_{2,3} = VALUE;

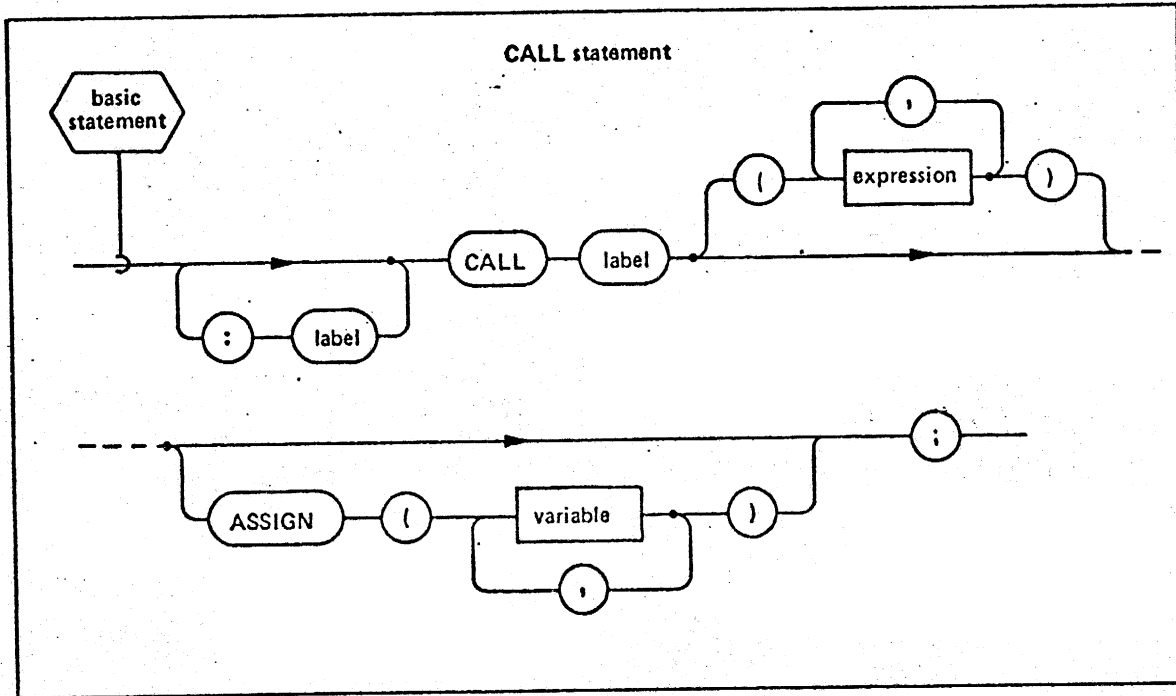
$\bar{V} = \bar{M} * \bar{N} / X^2;$

$\bar{D} = \text{VECTOR}_4(A, B, C, D);$ /* SHAPING FUNCTION */

7.4 The CALL Statement

The CALL statement is used to invoke execution of a procedure. Each <expression> is an "input argument", while each of the <variable>s is an "assign argument" whose values may be changed by the called procedure.

SYNTAX:

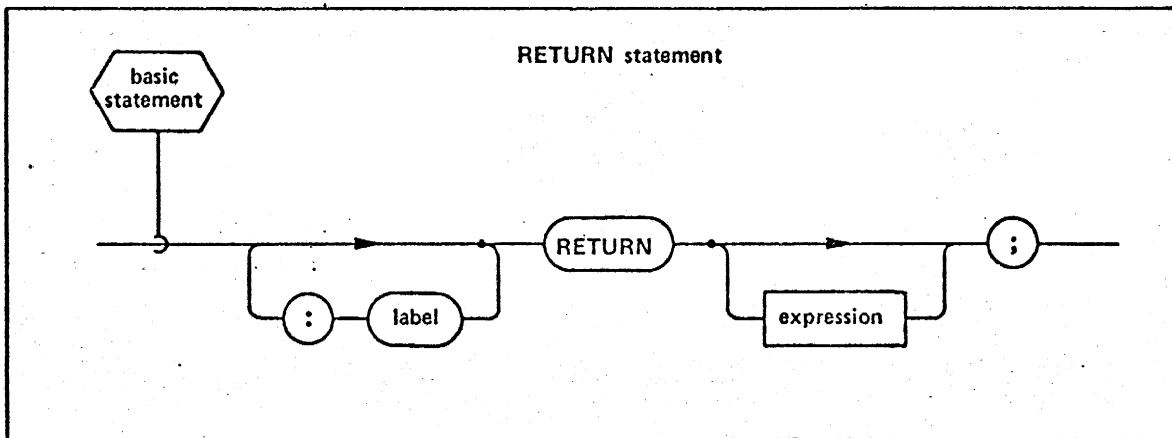


EXAMPLE: CALL EPSILON ASSIGN(KAPPA);
 ABLE: CALL GAMMA (ALPHA) ASSIGN(BETA,SIGMA);
 CALL PHI ($\bar{A}+\bar{B}, X^2, C$) ASSIGN($\dot{T}, \dot{U}, \bar{V}$);

7.5 The RETURN Statement

The RETURN statement is used to cause return of execution from a task, program, procedure, or function block. The <expression> may only appear in a <function block> RETURN statement.

SYNTAX:



```
EXAMPLE:  IF X>0 THEN RETURN;      /* PROCEDURE RETURN */
          DONE: RETURN;           /* PROCEDURE RETURN */

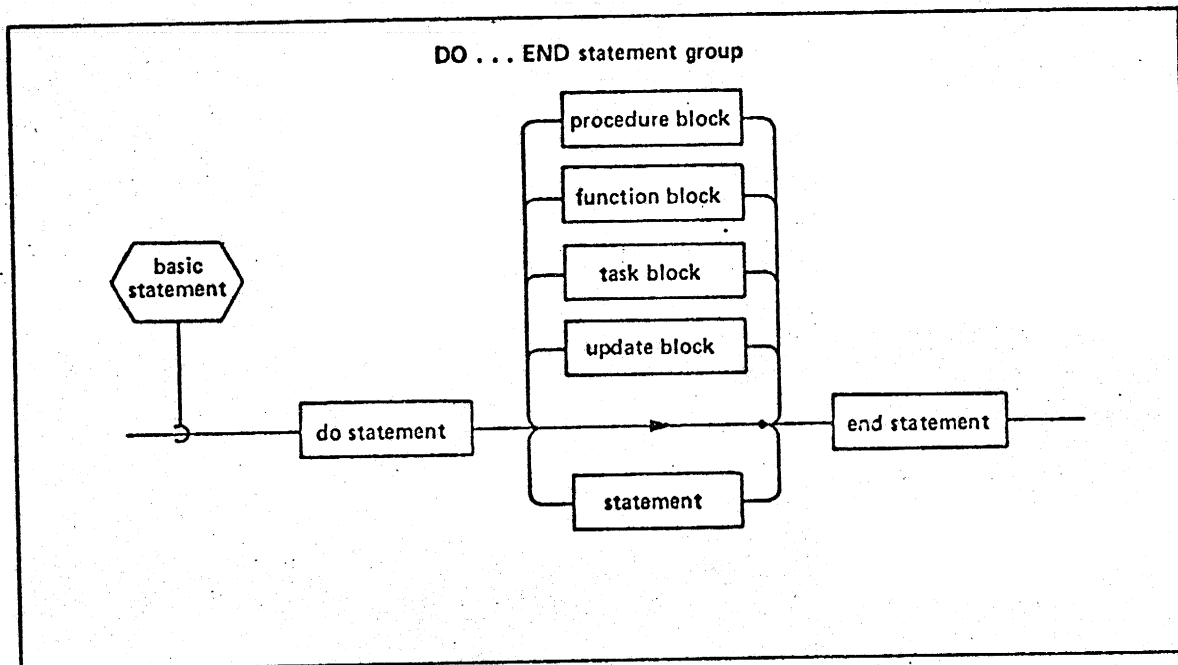
          IF X>0 THEN RETURN X3; /* FUNCTION RETURNS */
          ELSE RETURN -X3;

          DONE: RETURN  $\bar{A}+\bar{B}$ ; /* FUNCTION RETURN */
```

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.

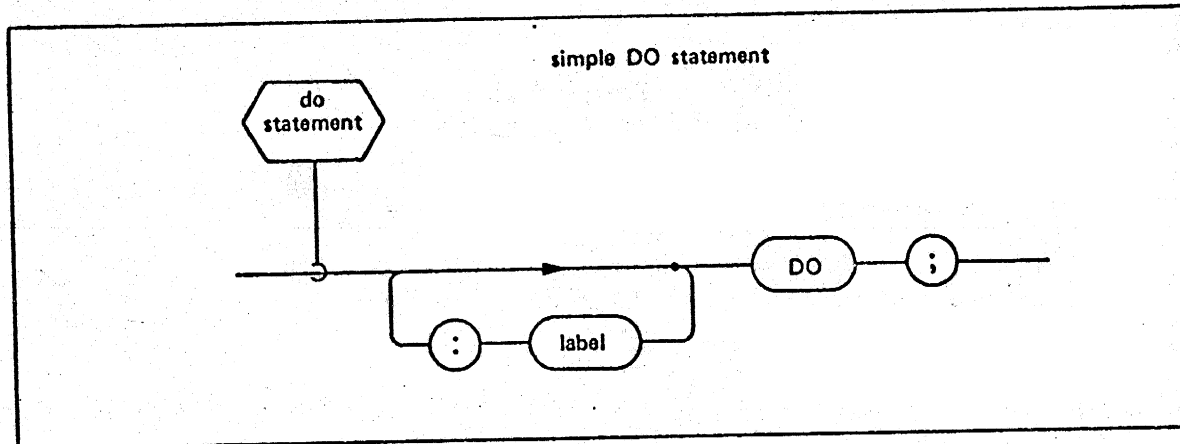
SYNTAX:



7.6.1 The Simple DO Statement

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

SYNTAX:

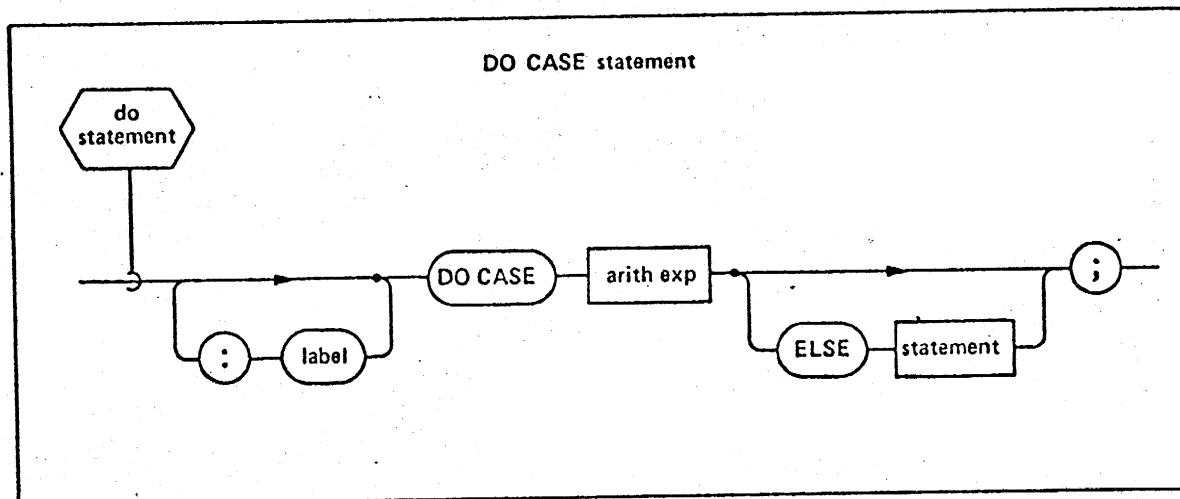


```
EXAMPLE:      ABLE: IF Z>SIG THEN
                DO;
                ALPHA=1;
                BETA=ALPHA/3;
                END;
            ELSE DO;
                Z=Z/SIG;
                SIG=SIG+1;
            END;
```

7.6.2 The DO CASE Statement

The DO CASE statement indicates that if the value of <arith exp> is an integer K, then of the following sequence of <statement>s comprising the group, the Kth statement of the group is executed. If K is either less than or equal to zero, or greater than the number of <statement>s in the group, then the <statement> following the ELSE keyword is executed; if there is no ELSE clause then a run time error occurs for such an invalid K-value.

SYNTAX:



EXAMPLES: ALPHA: DO CASE J-1;
 BETA=BETA+TAU;
 BETA=BETA/FACTOR+TAU;
 BETA=BETA/FACTOR;
 END;

 DO CASE N-3 ELSE GO TO ERROR1;
 SUM=VALUE+TAX;
 DIFF=TAX;
 DO;
 TOTAL=VALUE+TAX-DISCOUNT;
 CALL BILLER(VALUE);
 CALL SUMMARY(TAX);
 END;

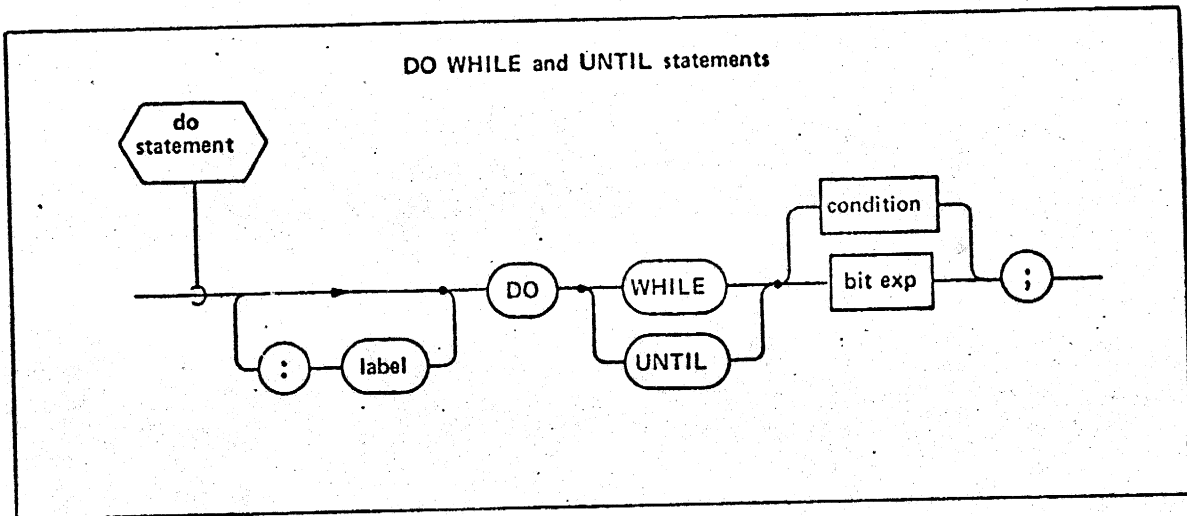
 END;
 ERROR1: IF VALUE>=0 THEN GO TO CONTINUE;

7.6.3 The DO WHILE and DO UNTIL Statements

The DO WHILE statement causes the group of <statement>s to be repeatedly executed until the value of <condition> or <bit exp> becomes false. The value is tested prior to each cycle of execution.

The DO UNTIL statement causes the group of <statement>s to be repeatedly executed until the value of <condition> or <bit exp> becomes true. The value is not tested prior to the first cycle of execution, but is tested before all subsequent cycles of execution.

SYNTAX:



```

EXAMPLES: DO WHILE I>0;
          J=0;
          ⋮
          VALUE=VALUE/I;
        END;

EQUIVALENTLY: DO UNTIL I<=0;
          J=0;
          ⋮
          VALUE=VALUE/I;
        END EQUIVALENTLY ;

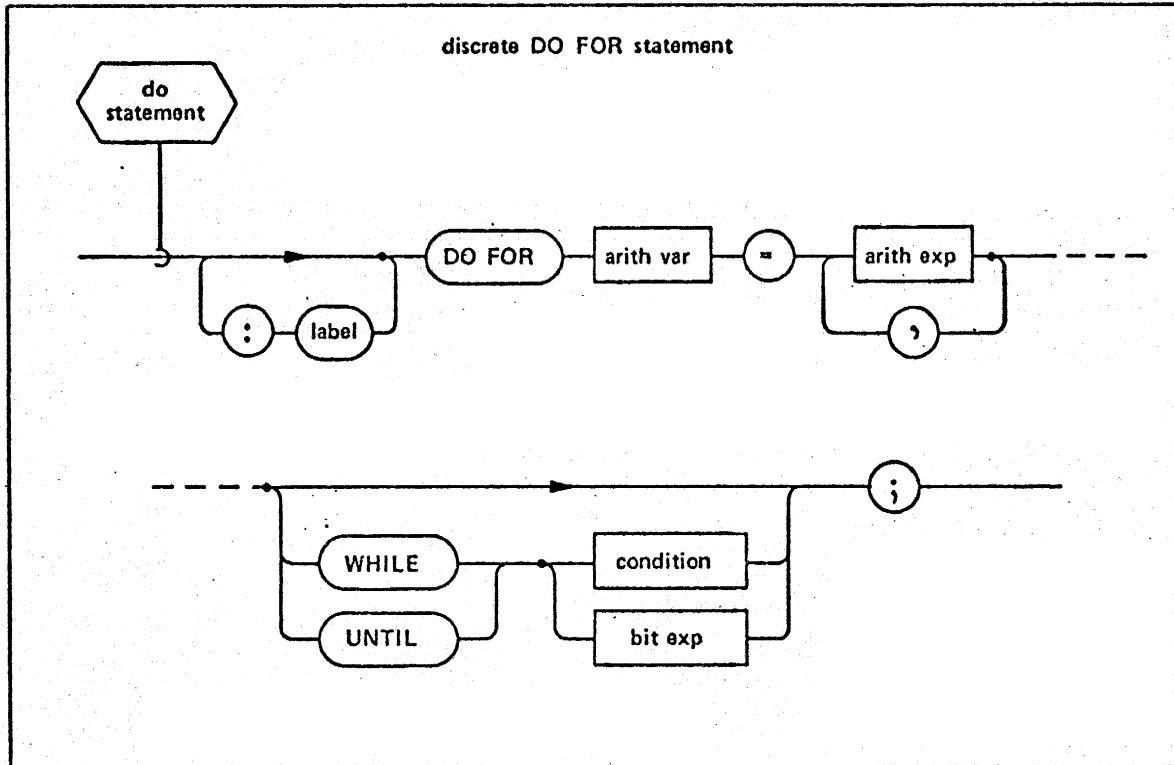
        DO WHILE A&(B|C);
          ⋮
        END;

```

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". Prior to each cycle of execution, the next <arith exp> in the list is evaluated and assigned to the loop variable. The presence of a WHILE or UNTIL clause is used to cause execution to be dependent on some condition being satisfied as in Section 7.6.3.

SYNTAX:

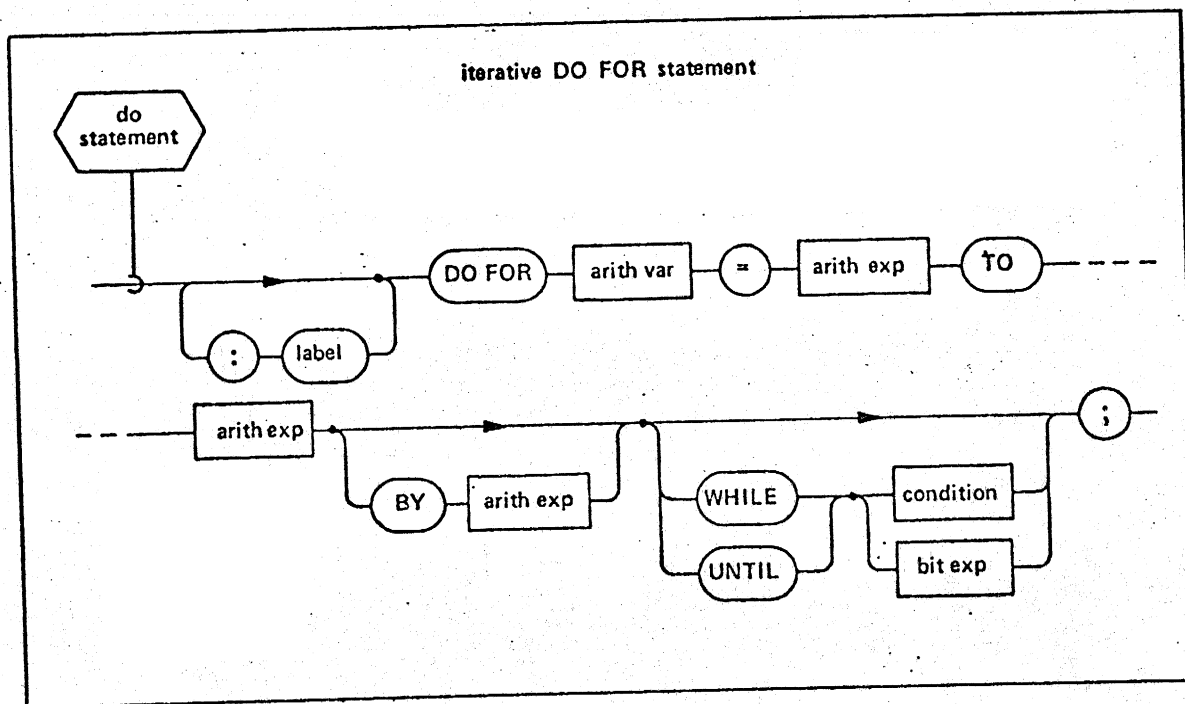


EXAMPLE: DO FOR I=10,20,30 WHILE J>0;
 NEWVAL=OLDVAL/I+INCRMT;
 :
 J+NEWVAL;
 END;

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 (the default value is 1).

SYNTAX:



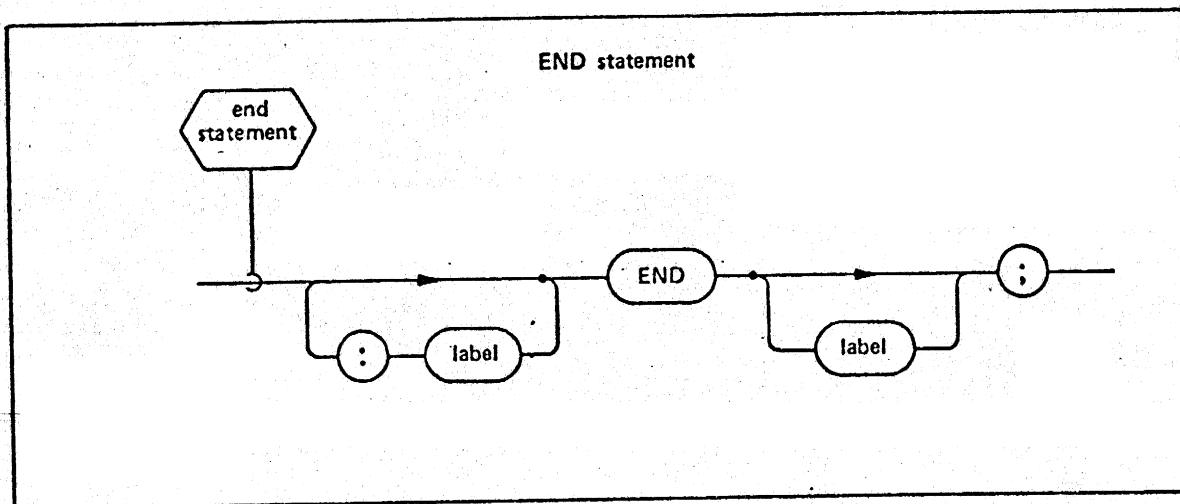
EXAMPLE:

```
EQUIVALENT_TO_LAST_EX: DO FOR I=10 TO 30 BY 10 UNTIL J<=0;  
    NEWVAL=OLDVAL/I+INCRMT;  
    ⋮  
    J=NEWVAL;  
END EQUIVALENT_TO_LAST_EX;  
  
DO FOR J=-30 TO 50 BY INCREMENT;  
    ⋮  
END;
```

7.6.6 The END Statement

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

SYNTAX:



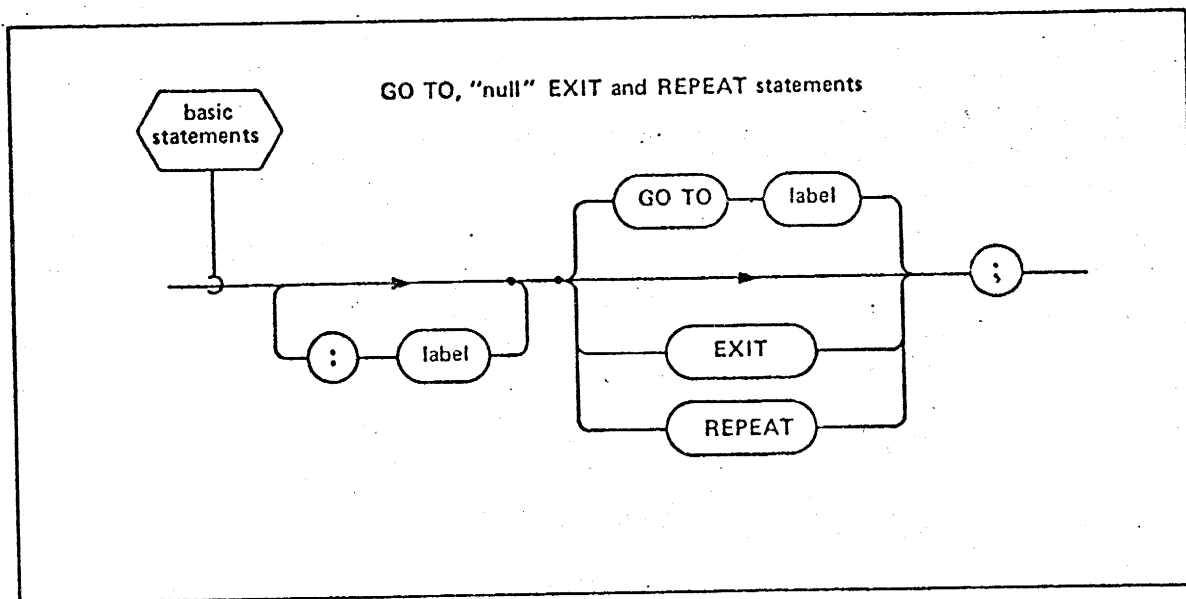
EXAMPLE:

```
LOOP: DO FOR...;  
    ⋮  
FINISH: END LOOP;
```

7.7 Other Basic Statements

- The GO TO <label> causes a branch in execution to an executable statement bearing the same <label>.
- The "null" statement has no effect at run time.
- The EXIT statement is legal only inside a DO...END group where it causes a branch to the first executable statement after the end of the DO...END group.
- The REPEAT statement is legal only inside a DO...END group opened with a DO FOR, DO WHILE or DO UNTIL statement. It causes immediate abandonment of the current cycle of execution of the innermost such group.

SYNTAX:



EXAMPLES:

```
DO FOR...;
  :
  ABLE: IF X>0 THEN EXIT;
        ELSE REPEAT;
        IF Y<10 THEN GO TO ABLE;
END;
```

8.0 REAL TIME CONTROL

HAL/S contains a comprehensive facility for creating a multi-processing job structure in a real time programming environment. At run time a Real Time Executive (RTE) controls the execution of processes held in a process queue. HAL/S contains statements which can 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 block 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

8.2 Timing Considerations

In the HAL/S system, the RTE accesses a clock measuring elapsed time ("RTE-clock" time). Time is measured in Machine Units (MU) whose correspondence with physical time

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

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

The SCHEDULE statement is used to request initiation of a program or task

- a) at a specific time (AT<arith exp>)
- b) in an incremental time (IN<arith exp>)
- c) on an event expression value of TRUE (ON<event exp>)

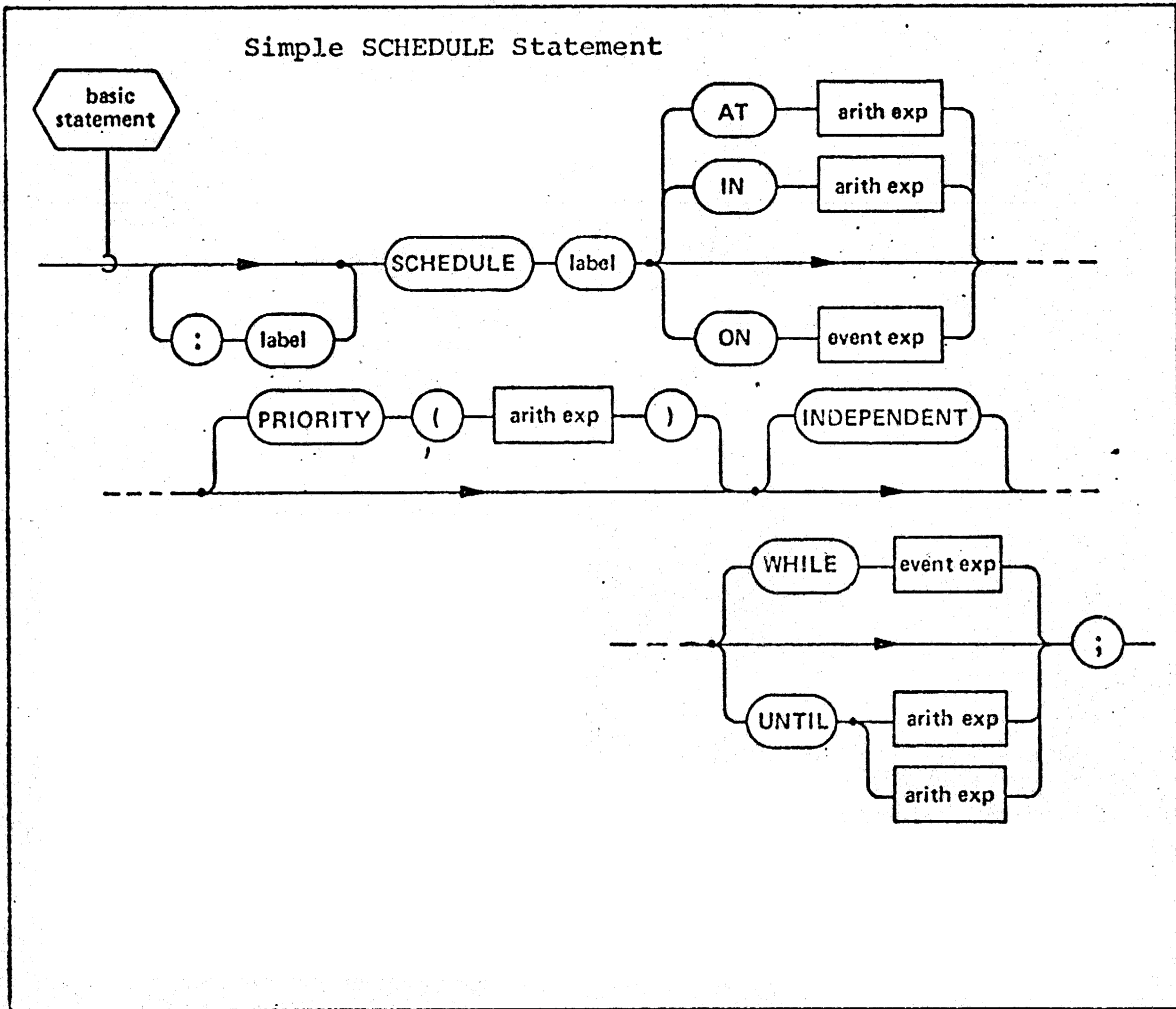
The initiation priority is explicitly set by use of the phrase PRIORITY (<arith exp>). If INDEPENDENT is specified, the scheduled program or task can continue in an active (executing) state even after the scheduling block has been terminated (although a task-son can never be independent of its program-father).

There are two forms of the SCHEDULE statement: the simple SCHEDULE statement and the cyclic SCHEDULE statement.

8.3.1 The Simple SCHEDULE Statement

The simple SCHEDULE statement initiates a program or task only once. Initiation will not occur if the value of the <arith exp> to the right of the keyword UNTIL is less than the RTE-clock time specified for initiation. Similarly, initiation will not occur if the value of the <event exp> to the right of the keyword WHILE is FALSE upon execution of the SCHEDULE statement or at any time before the process is initiated. The clause UNTIL <event exp> has no effect on a simple SCHEDULE statement.

SYNTAX:



EXAMPLES: SCHEDULE IOTA;
 SCHEDULE RADAR ON R_RUPT OR C_RUPT PRIORITY(HIGH);
 SCHEDULE TRACK AT 15;
 SCHEDULE TRACK ON TRACK_FLAG UNTIL 15;

8.3.2 The Cyclic SCHEDULE Statement

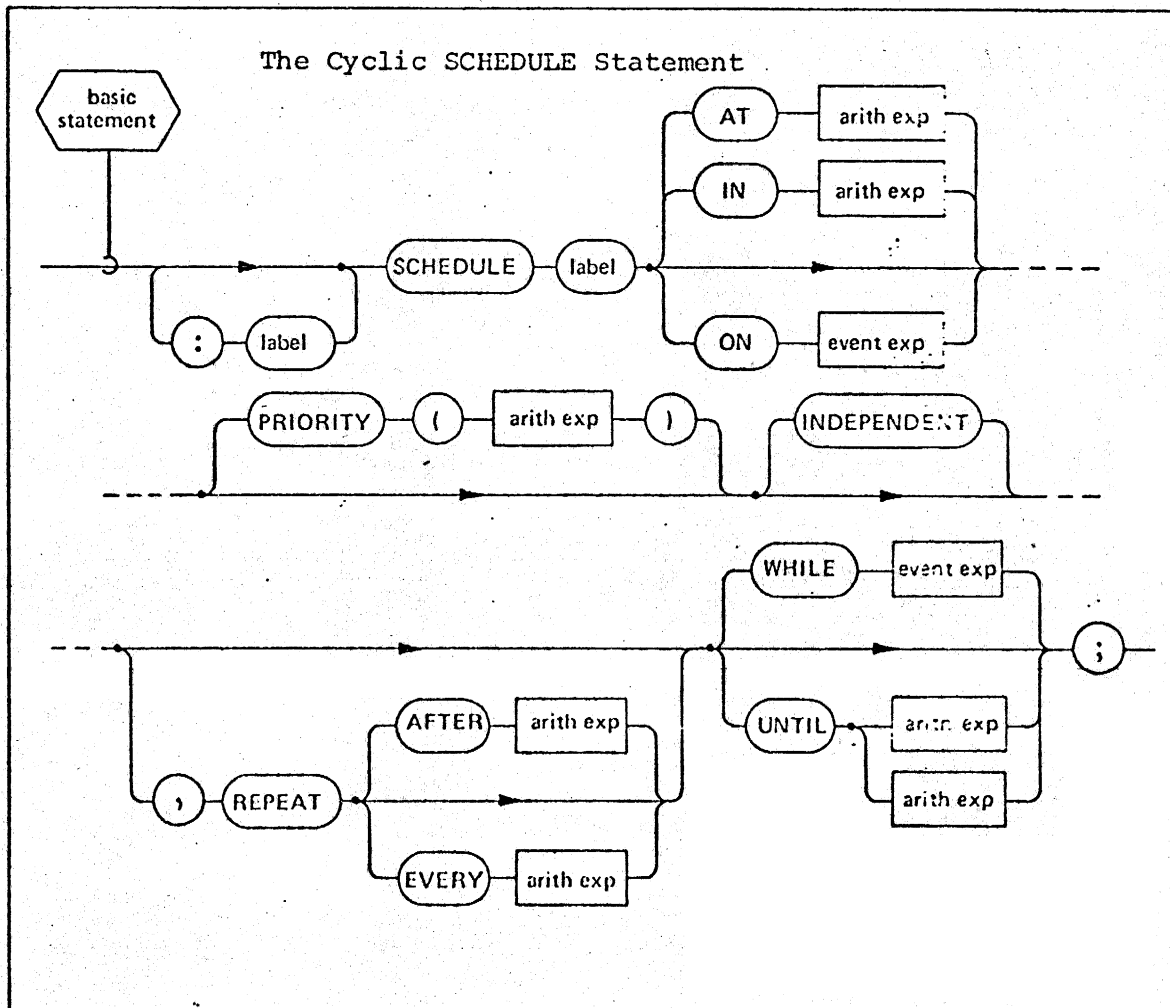
The cyclic SCHEDULE statement contains a REPEAT phrase which causes the RTE to cyclically execute the process as long as one of the following holds:

- a) UNTIL <arith exp> is greater than the RTE-clock time.
- b) UNTIL <event exp> is evaluated to be FALSE.
- c) WHILE <event exp> is evaluated to be TRUE.

These evaluations are made prior to each cycle, but UNTIL <event exp> is not evaluated until the second and subsequent cycles.

To cause a fixed RTE-clock time delay between the completion of the previous and the beginning of the next cycle, the qualifier AFTER <arith exp> is used. 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.

SYNTAX:



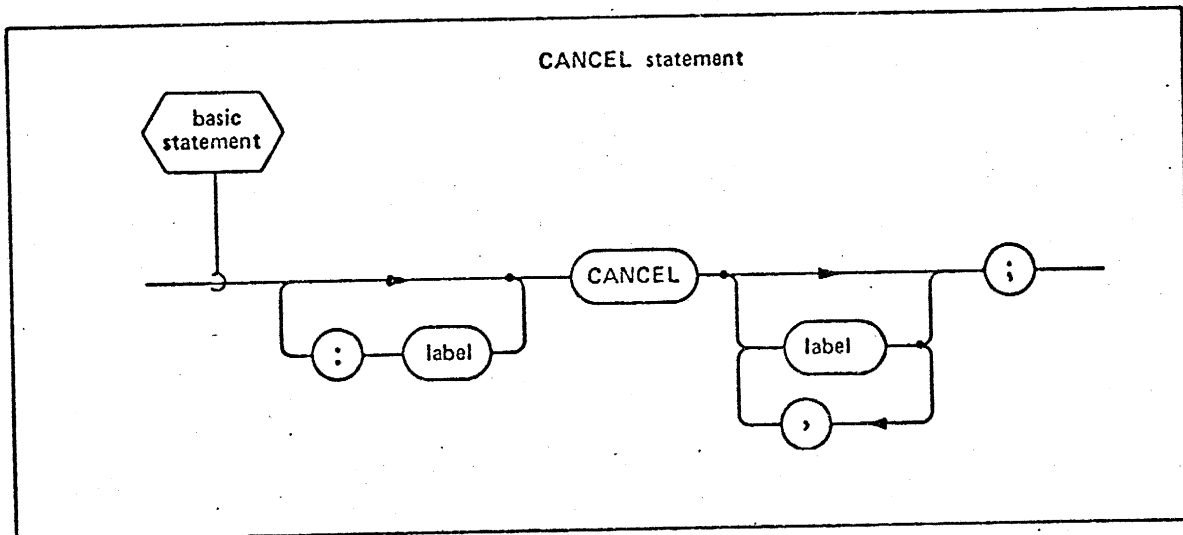
EXAMPLES: SCHEDULE DELTA INDEPENDENT, REPEAT EVERY 15.9
UNTIL 75.9;

SCHEDULE STEERING AT TIG-5 PRIORITY(6), REPEAT
EVERY 2 WHILE ENG_ON;

8.4 The CANCEL Statement

When a CANCEL statement is used, if the process is non-cyclic no action is taken. If the process is cyclic, then the process is cancelled at the end of the current cycle of execution after possibly waiting for any dependent sons to terminate.

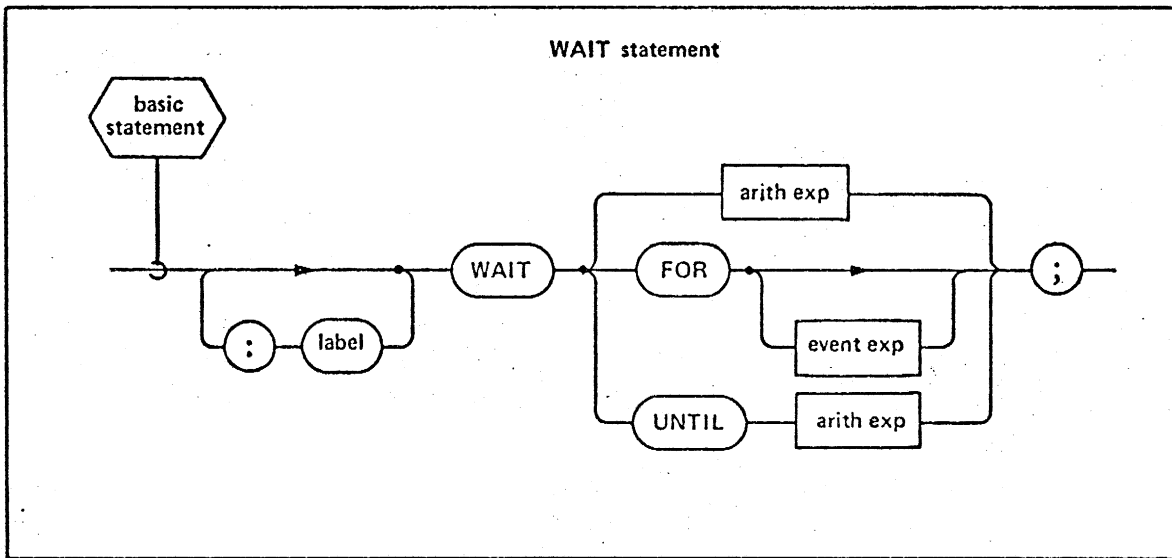
SYNTAX:



EXAMPLE: CLEAN_UP: CANCEL ETA, NU;
IF A&B&C THEN CANCEL TRACK_JOB;

- c) upon termination of all dependent sons: WAIT FOR
- d) upon a TRUE value of an event expression evaluated at each "event change point" (see Section 8.8):
WAIT FOR <event exp>.

SYNTAX:

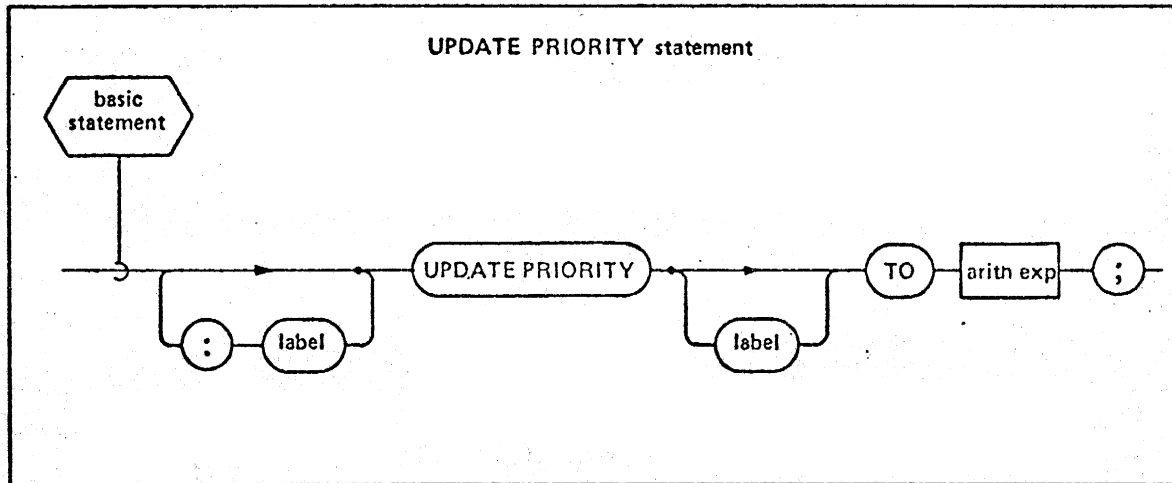


EXAMPLES: NOW: WAIT UNTIL T+7.5;
 WAIT 5;
 WAIT FOR ABLE;
 WAIT FOR; /* TERMINATION OF DEPENDENT SONS */
 WAIT FOR ~ ABLE OR BAKER;

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 to <arith exp> by means of the UPDATE PRIORITY statement. UPDATE PRIORITY with no <label> specification is used to change the priority of the process executing the UPDATE PRIORITY statement.

SYNTAX:



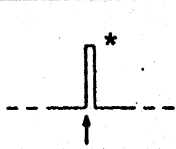
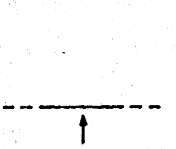
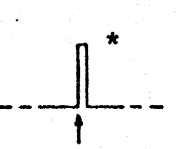


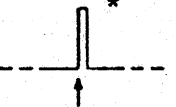



EXAMPLE: UPDATE PRIORITY GAMMA TO 10;
 UPDATE PRIORITY TO K+5;

8.8 Events and the SIGNAL Statement

At any instant of time the RTE may be viewed as having knowledge of all existing events whenever the value of an event changes, the RTE senses this "event change point" and may in turn perform the evaluation of pending <event exp>s.

The value of an event variable can be changed by the use of the SIGNAL statement. Depending upon the implementation and the available computer hardware, event variables shall also respond to the external environment (either by activation of the SIGNAL statement or by special operating system provision).

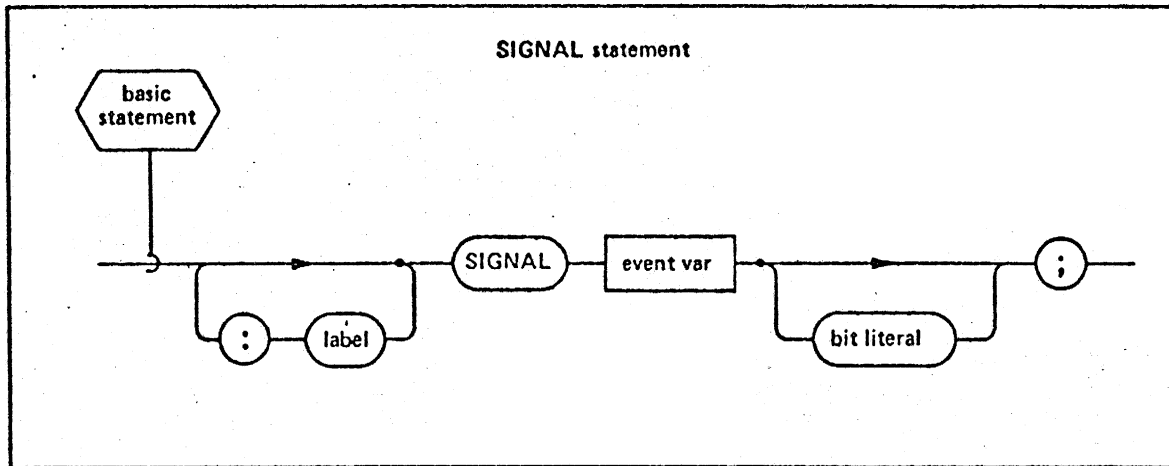
The operation of the SIGNAL statement is summarized as follows:

type of event and initial value	type of SIGNAL statement		
	SIGNAL .. ON	SIGNAL .. OFF	SIGNAL ..
unlatched, FALSE			
latched, FALSE			
latched, TRUE			

* The <event var> is TRUE for a period of time invisible to the HAL/S user but long enough to be detectable by the RTE.

NOTE: T ⇔ TRUE
F ⇔ FALSE

SYNTAX:



EXAMPLE: SIGNAL IOTA ON;

8.9 Process-Events

Any program or task block may have associated with it a so-called "process-event" of the same name. This process-event behaves in every way like a latched event except that it may not appear in 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 be declared with the LOCKED attribute. 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 "referenced" 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.

9.0 ERROR RECOVERY AND CONTROL

References to so-called 'run time errors' have been made elsewhere in this document. 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 positive integer, called the "error code" of that error. 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.

At run time an Error Recovery Executive (ERE) senses errors, both system-defined and user-defined, and determines what course of action to take. 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 simulation system-defined error occurrences.

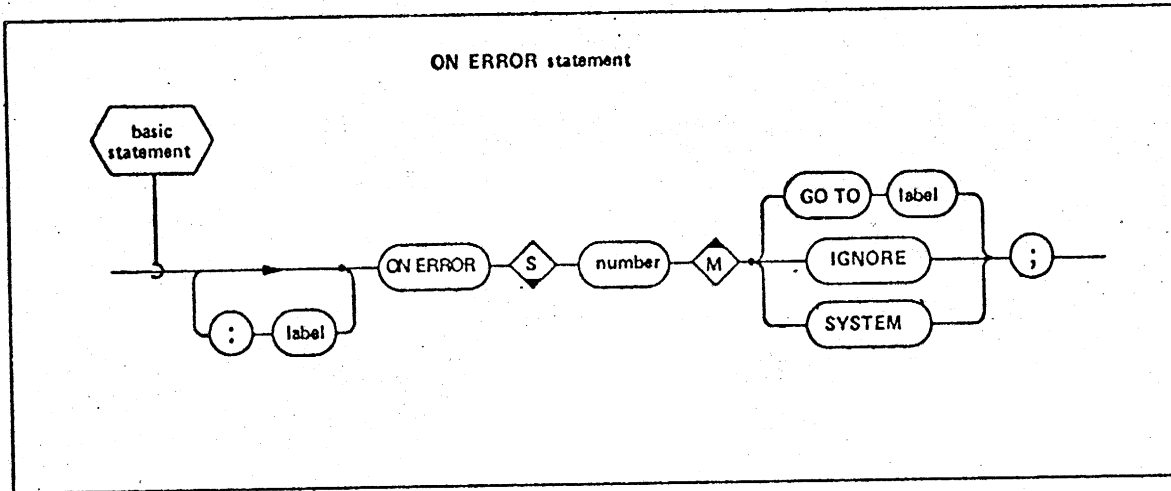
9.1 The ON ERROR Statement

The ON ERROR statement is used to modify the action of the error defined by <number> prevailing in the current program, task, procedure, function or update block, in the following manner:

- a) the GO TO <label> clause causes the ERE to branch to <label> when the specified error occurs.
- b) the IGNORE clause allows execution as if the error had not occurred.
- c) the SYSTEM clause causes the ERE to take standard system recovery action.

GO TO and/or IGNORE action may not be permitted for some errors.

SYNTAX:



EXAMPLES:

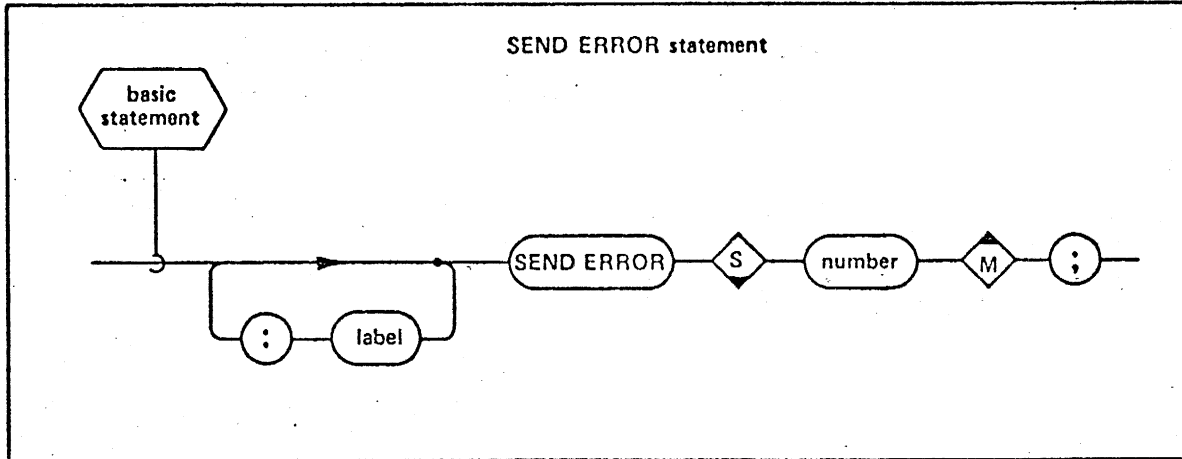
ERRONEOUS: ON ERROR₅ IGNORE;

ON ERROR₂₇ GO TO RECOVERY;

9.2 The SEND ERROR Statement

The SEND ERROR statement is used to announce the error condition defined by <number> to the ERE. If <number> corresponds to a system defined error, then that error is said to be simulated by the ERE. The action of the ERE is dictated by the error environment prevailing at the time of execution of the SEND ERROR statement.

SYNTAX:



EXAMPLE: TEST_CONDITION: IF ERR_FLAG THEN SEND ERROR₁₅;

10.0 INPUT/OUTPUT STATEMENTS

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

All HAL/S I/O is directed to one of a number of input/output "channels". These channels are the means 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.

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 being 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 a read or write "device mechanism", which actually performs the data transfer, across the file. 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.

10.1.1 The READ and READALL Statements

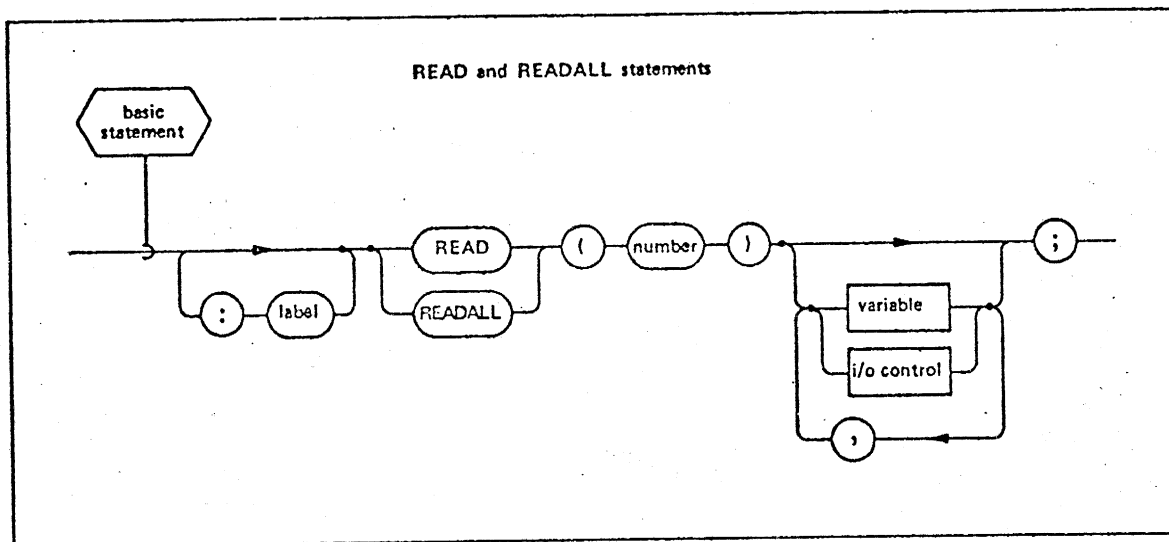
The READ statement is used for the sequential input of data in a standard external format. Each field of contiguous characters separated by commas, semicolons or blanks is converted to an appropriate HAL/S data value assigned to

the <variable>. A semicolon field separator terminates the READ statement with any unassigned <variable>s left unchanged.

The READALL statement is used for the sequential input of unconverted, arbitrary character string images to be assigned to any character variable and/or structure containing only character strings.

<number> is any legal I/O channel number. <i/o control> is an optional control function used to position the device mechanism explicitly (see Section 10.1.3).

SYNTAX:

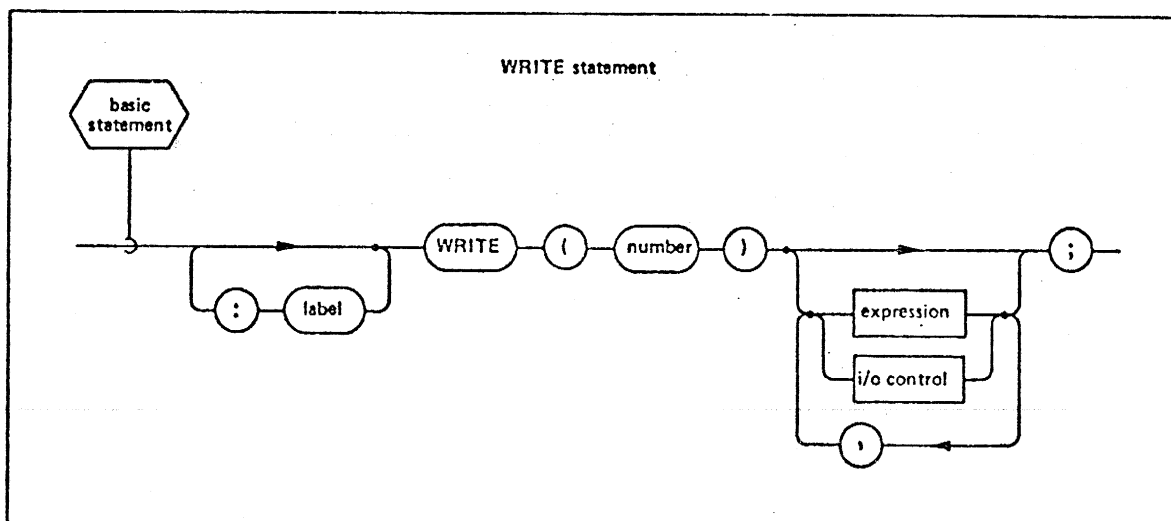


EXAMPLES: READ (CARDS) A,B,C,D,[E],[F];
 READ (CARDS) COLUMN(20),A,B,
 SKIP(1), COLUMN(20),C,D,
 SKIP(1), COLUMN(20),E,F,
 :
 etc.
 READALL(CARD) C, COLUMN(40),D;
 READ (CARDS) A, TAB(40),C;

10.1.2 The WRITE Statement

The sequential output of data in standard external format on the channel specified by <number> is accomplished by using the WRITE statement. Unless overridden by an <i/o control>, between the transmission of two consecutive elements, the device mechanism is moved to the right by a fixed implementation dependent number of columns.

SYNTAX:



EXAMPLES:

```
WRITE(LISTING)A,B,C*,D,[E],[F];
WRITE(LISTING)A,TAB(10),B,COLUMN(50),C;
WRITE(6) ALPHA, SKIP(2), BETA;
```

10.1.3 I/O Control Functions

An I/O control function in a READ, READALL or WRITE statement causes the explicit movement of the device mechanism. If the value of K is specified by the signed integer value of <arith exp>, then:

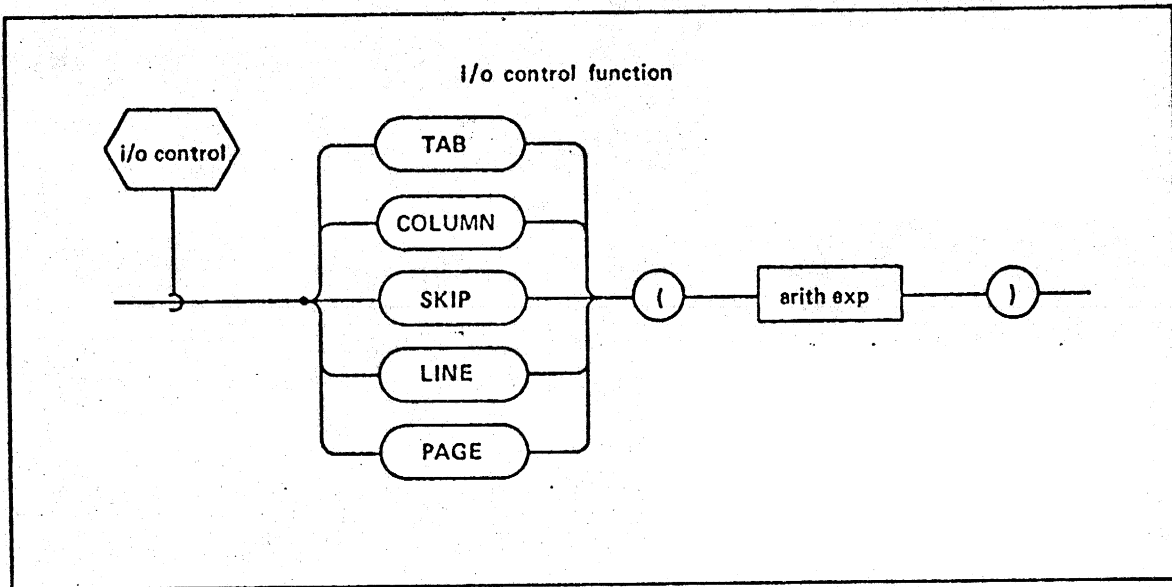
- TAB(K) specifies relative movement of the device mechanism across the current line. Motion is to the right by K character positions for positive K.
- COLUMN(K) specifies absolute movement of the device mechanism to column K of the current line.
- SKIP(K) specifies line movement of K lines relative to the current line of the file. Subject to implementation restrictions, backward movement is indicated by negative values of K.
- LINE(K) specifies line movement to the specified line number:

paged files - LINE(K) advances the file unconditionally, advancing to line K of the next page if K is less than the current line number.

unpaged files - LINE(K) positions the device mechanism at some absolute line number in the file.

- PAGE(K) is applicable to paged files only and specifies movement K pages forward relative to the current page. Subject to implementation restrictions, backward movement is indicated by negative values of K. In either case, the line value relative to the beginning of the page is unchanged.

SYNTAX:



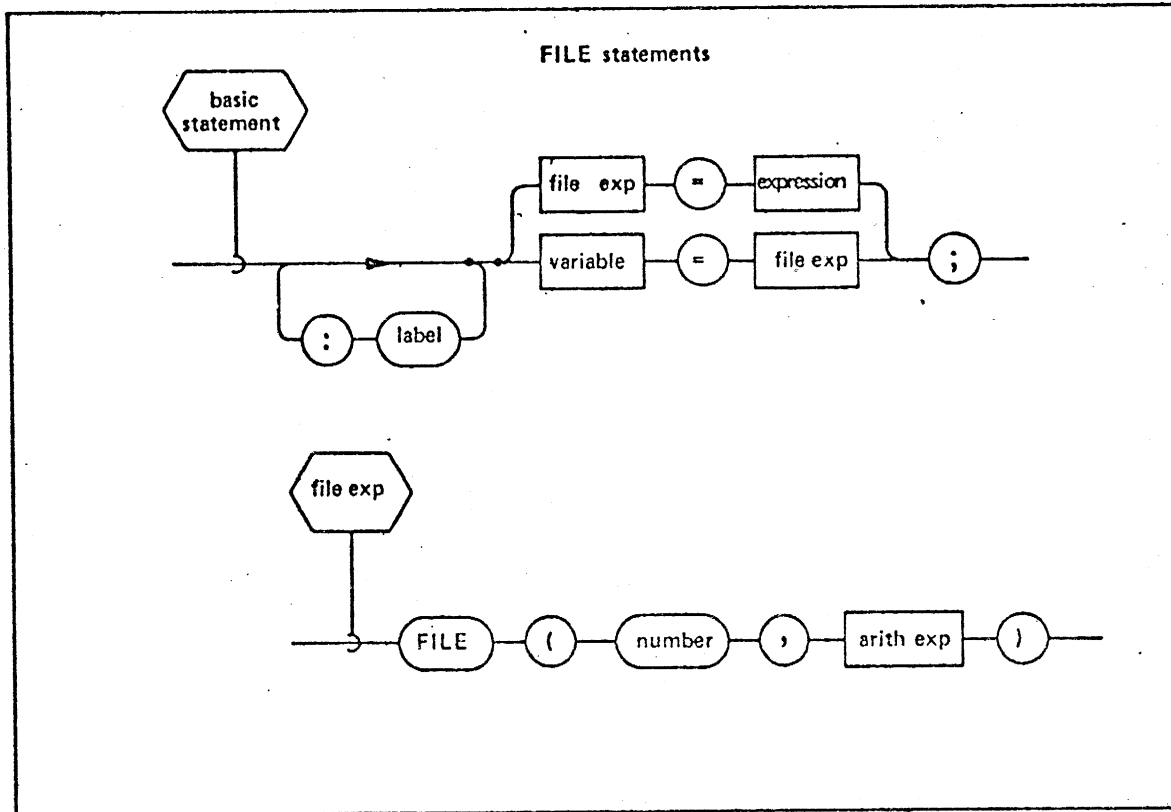
10.2 Random Access I/O - The FILE Statement

Individual records, specified by the <file exp> "record address", on a file may be written, retrieved, or updated via the FILE statement. <number> is a legal random access channel number. <arith exp> is any unarrayed integer or scalar expression.

INPUT: When <file exp> is on the right-hand side of the assignment, the statement is an input FILE statement where <variable> is any variable usable in an assignment context.

OUTPUT: When <file exp> is on the left-hand side, the statement is an output FILE statement where there are no semantic restrictions on <expression>.

SYNTAX:



EXAMPLES:

FILE(3,J+2) = ALPHA₁ TO 1000;

FILE(TAPE,I) = [A]; /* TAPE IS AN INTEGER LITERAL */

{B} = FILE(DISC,A_I); /* DISC IS AN INTEGER LITERAL */

APPENDIX A.
HAL/S Keywords
(not including built-in functions)

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

APPENDIX B.

HAL/S Built-In Functions*

A. String Functions (Bit or Character String Arguments)

INDEX (string, config)
LENGTH [applies to character-strings only]
LJUST
RJUST (character-string, length)

B. Arithmetic Functions (Integer or scalar arguments)

ABS
CEILING
FLOOR
ROUND
SIGNUM
SIGN
TRUNCATE
MOD (numerator, denominator)
DIV
REMAINDER
MAX
MIN
ODD

C. Mathematical Functions (Integer or scalar arguments)

ARCCOS
ARCSIN
ARCTAN
COS
SIN
TAN
EXP
LOG
SQRT
ARCCOSH
ARCSINH
ARCTANH
COSH
SINH
TANH

* Note: This list is typical; the actual list in force is implementation-dependent. All functions require single arguments except where that or more arguments are shown in parentheses following the name.

D. Matrix-Vector Functions

ABVAL
DET
INVERSE
TRACE
TRANSPOSE
UNIT

E. Linear Array Functions

SUM
PROD
MAX
MIN
SIZE

F. Miscellaneous Functions

RANDOM
RANDOMG
DATE
RUNTIME
CLOCKTIME
PRIO

APPENDIX C.

Summary of HAL/S Operations

The following tables summarize the allowable operations between two operands. In most cases the valid result-type (or an error) and any implied data conversions are indicated within the boxes.

Operation Prefix :

$\begin{matrix} P \\ \{Q\} \end{matrix} op_2$

P: { \pm }

Q: NOT (\neg)

Op_2	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
	P INTEGER	P SCALAR	P VECTOR	P MATRIX	Q BIT STRING	

Table C-1

Operation Addition & Subtract :

$$Op_1 \pm Op_2$$

$Op_1 \backslash Op_2$	INTEGER	SCALAR	VECTOR	MATRIX
INTEGER	INTEGER	SCALAR I→S	ERROR	ERROR
SCALAR	SCALAR I→S	SCALAR	ERROR	ERROR
VECTOR	ERROR	ERROR	VECTOR d	ERROR
MATRIX	ERROR	ERROR	ERROR	MATRIX d

I→S = conversion of integer to scalar

d = dimension check

Table C-2

Operation Multiplication:

Op₁ Op₂

OPERAND ₂ \ OPERAND ₁	INTEGER	SCALAR	VECTOR	MATRIX
INTEGER	INTEGER	SCALAR I+S	VECTOR I+S	MATRIX I+S
SCALAR	SCALAR I+S	SCALAR	VECTOR	MATRIX
VECTOR	VECTOR I+S	VECTOR	MATRIX (1) SCALAR (2) VECTOR (3)	VECTOR d
MATRIX	MATRIX I+S	MATRIX	VECTOR d	MATRIX d

- Notes:
- (1) Vector outer product $\bar{V} \bar{V}$
 - (2) Vector DOT product $\bar{V} \cdot \bar{V} (d)$
 - (3) Vector cross product $\bar{V} * \bar{V} (d, \text{restricted to 3-element vectors})$
- d: dimension check
I+S: integer to scalar conversion

Table C-3

Operation Division :

Op_1/Op_2

$Op_1 \backslash Op_2$	INTEGER	SCALAR	VECTOR	MATRIX
INTEGER	SCALAR I+S	SCALAR I+S	ERROR	ERROR
SCALAR	SCALAR I+S	SCALAR	ERROR	ERROR
VECTOR	VECTOR I+S	VECTOR	ERROR	ERROR
MATRIX	MATRIX I+S	MATRIX	ERROR	ERROR

I+S: integer to scalar conversion

Table C-4

Operation Exponentiation :

$$Op_1 ** Op_2$$

Op ₁ \ Op ₂	INTEGER	SCALAR	VECTOR	MATRIX
INTEGER	SCALAR (1) .. I→S	SCALAR (1) I→S	ERROR	ERROR
SCALAR	SCALAR I→S	SCALAR	ERROR	ERROR
VECTOR	ERROR	ERROR	ERROR	ERROR
MATRIX	MATRIX	MATRIX S→I	ERROR	ERROR

Note (1) Result is Integer if Op₂ is a whole number literal ≥
(no I→S)

Table C-5

Operation Relational :

$Op_1 \begin{Bmatrix} P \\ Q \end{Bmatrix} Op_2$

P: =, ≠

Q: =, ≠, >, <, <=, >=, ≠, <, >

$Op_1 \backslash Op_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	Q	Q I→S	ERROR	ERROR	ERROR	ERROR
SCALAR	Q I→S	Q	ERROR	ERROR	ERROR	ERROR
VECTOR	ERROR	ERROR	P	ERROR	ERROR	ERROR
MATRIX	ERROR	ERROR	ERROR	P	ERROR	ERROR
BIT STRING	ERROR	ERROR	ERROR	ERROR	P ⁽¹⁾	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	P ⁽²⁾

Special: <structure>P<structure>

Notes: (1) Operand padded on left to equalize lengths if necessary.

(2) Operand padded on right to equalize lengths if necessary.

Table C-6

Operation String : $OP_1 \left\{ \begin{matrix} P \\ Q \end{matrix} \right\} OP_2$ P: ||
 Q: ||, AND, OR

OPERAND ₂ OPERAND ₁	INTEGER	SCALAR	BIT STRING	CHARACTER STRING
INTEGER	P I+C I+C	P I+C S+C	ERROR	P CHARACTER I+C
SCALAR	P S+C I+C	P S+C S+C	ERROR	P CHARACTER
BIT STRING	ERROR	ERROR	Q BIT STRING	ERROR
CHARACTER STRING	P CHARACTER I+C	P CHARACTER S+C	ERROR	P CHARACTER

I+C: Conversion from integer to character
 S+C: Conversion from scalar to character

Table C-7

APPENDIX D.

Conversion Functions

1. Summary of conversion function results when unsubscripted with single (unrepeated) argument; e.g. SCALAR(\bar{V}).

A. INTEGER, SCALAR, BIT, CHARACTER

arguments \	X	$[X]_a$	$[X]_{a,b}$	\bar{V}_ℓ	$[\bar{V}]_{a:\ell}$	$\overset{*}{M}_{m,n}$	$[\overset{*}{M}]_{a:m,n}$
INTEGER	Y	$[Y]_a$	$[Y]_{a,b}$	$[Y]_\ell$	$[Y]_{ax\ell}$	$[Y]_{mxn}$	$[Y]_{axmxn}$
SCALAR	Y	$[Y]_a$	$[Y]_{a,b}$	$[Y]_\ell$	$[Y]_{ax\ell}$	$[Y]_{mxn}$	$[Y]_{axmxn}$
BIT	Y	$[Y]_a$	$[Y]_{a,b}$	Error	Error	Error	Error
CHARACTER	Y	$[Y]_a$	$[Y]_{a,b}$	Error	Error	Error	Error

X may be integer, scalar, bit or character type.

Y indicates same data type as function.

a,b indicate array shape (in general, may be a,b,c, etc.).

ℓ indicates vector length.

m,n indicate matrix row and column dimensions respectively.

x (lower case x) indicates "by" as in 'axmxn' = "a by m by n"

B. VECTOR, MATRIX

(arguments may be of integer and scalar type only)

An unsubscripted VECTOR always produces a 3-vector; therefore the number of elements in the argument must be exactly 3.

An unsubscripted MATRIX always produces a 3-by-3 matrix; therefore the number of elements in the argument must be exactly 9.

2. Summary of Conversion Function 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	✓	✓			✓	✓

APPENDIX E.

Sample Program Listing

The following program was written in HAL/360, and is included only as an example of the main features of the HAL language.

HAL COMPILER PHASE 1 -- VERSION OF AUGUST 24, 1972. CLOCK TIME = 18:58:37.08.

TODAY IS OCTOBER 1, 1972. CLOCK TIME = 12:31:44.20.

STMT	SOURCE	CURRENT SCOPE
1 MI	CONIC_STATE_EXTRAP:	CONIC_STATE_EXTRAP
1 MI	PROGRAM;	CONIC_STATE_EXTRAP
2 MI	DECLARE UNIVERSAL_KEPLER PROCEDURE,	CONIC_STATE_EXTRAP
2 MI	SECANT_ITER PROCEDURE,	CONIC_STATE_EXTRAP
2 MI	EXTRAP_STATE PROCEDURE;	CONIC_STATE_EXTRAP
3 MI	DECLARE PI CONSTANT(3.14159);	CONIC_STATE_EXTRAP
4 MI	DECLARE MU CONSTANT(1234);	CONIC_STATE_EXTRAP
5 MI	DECLARE DELT, X, DELT_CPRIME, X_CPRIME;	CONIC_STATE_EXTRAP
6 MI	DECLARE VECTOR,	CONIC_STATE_EXTRAP
6 MI	R0, V0, R, V;	CONIC_STATE_EXTRAP
7 MI	DECLARE DELT_C, X_C;	CONIC_STATE_EXTRAP
8 MI	UNIVERSAL_KEPLER:	UNIVERSAL_KEPLER
8 MI	PROCEDURE(C1, C2, X, XI, ROMAG) ASSIGN(DELTA_C, S_OF_XI, C_OF_XI);	UNIVERSAL_KEPLER
9 MI	DECLARE C1, C2, X, XI, ROMAG;	UNIVERSAL_KEPLER
10 MI	DECLARE S_OF_XI, C_OF_XI, DELTA_C;	UNIVERSAL_KEPLER
11 MI	ONE = 1;	UNIVERSAL_KEPLER
12 MI	CLOSE UNIVERSAL_KEPLER;	UNIVERSAL_KEPLER

B L O C K S U M M A R Y

IMPLICITLY DECLARED VARIABLES:
ONE

TEXT

SOURCE

CURRENT SCOPE

13 M1 SECANT_ITER:

| SECANT_ITER

13 M1 PROCEDURE(DELT_C, DELT_CPPIME, T_ERR, X) ASSIGN(XMIN, YMAX, DELX, S):

| SECANT_ITER

14 M1 DECLARE T_ERR, DELT_C, DELT_CPPIME, X:

| SECANT_ITER

15 M1 DECLARE XMIN, XMAX, DELX, S:

| SECANT_ITER

16 M1 TWO = 2:

| SECANT_ITER

17 M1 CLOSE SECANT_ITER:

| SECANT_ITER

B L O C K S U M M A R Y

IMPLICITLY DECLARED VARIABLES:

TWO

STATE SOURCE

	CURRENT SCOPE
18 *1 EXTRAP_STATE:	EXTRAP_STATE
18 *1 PROCEDURE(R0, V0, X, XI, S_OF_XI, C_OF_XI, DELT_C) ASSIGN(R, V):	EXTRAP_STATE
19 *1 DECLARE XI, S_OF_XI, C_OF_XI, X, DELT_C:	EXTRAP_STATE
20 *1 DECLARE VFCTOR,	EXTRAP_STATE
20 *1 R, V, R0, V0:	EXTRAP_STATE
21 *1 THREE = 3:	EXTRAP_STATE
22 *1 CLOSE EXTRAP_STATE;	EXTRAP_STATE

B L O C K S U M M A R Y

IMPLICITLY DECLARED VARIABLES:
THREE

STMT	SOURCE	CURRENT SCOPE
23 *1	KEPLER_ROUTINE:	KEPLER_ROUTINE
23 *1	DECL (R0, V0) ASSIGN(R, V, X_C, DELT, Y, DELT_CPRIME, X_CPRIME, DELT_C):	KEPLER_ROUTINE
24 *1	DECLARE DELT, DELT_CPRIME, X_CPRIME, X;	KEPLER_ROUTINE
25 *1	DECLARE VECTOR,	KEPLER_ROUTINE
25 *1	R0, V0, P, V, I_R;	KEPLER_ROUTINE
26 *1	DECLARE DELTMAX CONSTANT(345);	KEPLER_ROUTINE
27 *1	DECLARE EPS_T CONSTANT(22);	KEPLER_ROUTINE
28 *1	DECLARE EPS_X CONSTANT(33);	KEPLER_ROUTINE
29 *1	DECLARE I_MAX CONSTANT(12);	KEPLER_ROUTINE
30 *1	DECLARE BIT, LOOPING;	KEPLER_ROUTINE
31 *1	E1 ROMAG = ABVAL(R0);	KEPLER_ROUTINE
32 *1	E1 I_P = UNIT(R0);	KEPLER_ROUTINE
33 *1	E1 C1 = R0 . V0 / SORT(MU);	KEPLER_ROUTINE
34 *1	E1 C2 = ROMAG V0 . V0 / M1 - 1;	KEPLER_ROUTINE
35 *1	ALPHA = (1 - C2) / ROMAG;	KEPLER_ROUTINE
36 *1	IF ALPHA < 0 THEN	KEPLER_ROUTINE
36 *1	XMAX = SORT(-50 / ALPHA);	KEPLER_ROUTINE
37 *1	ELSE	KEPLER_ROUTINE
37 *1	DO:	KEPLER_ROUTINE
38 *1	XMAX = 2 PI / SORT(ALPHA);	KEPLER_ROUTINE
39 *1	P = 2 PI / ALPHA SORT(ALPHA MU);	KEPLER_ROUTINE
40 *1	IF P < DELTMAX THEN	KEPLER_ROUTINE
40 *1	DO WHILE ABS(DELT) >= P;	KEPLER_ROUTINE
41 *1	DELT = DELT - SIGN(DELT) P;	KEPLER_ROUTINE
42 *1	END;	KEPLER_ROUTINE
43 *1	END;	KEPLER_ROUTINE

STATE	SOURCE	CURRENT SCOPE
43 *	IF X SIGN(DELT) <= 0 OR (ABS(X) - XMAX) >= 0 THEN	KEPLER_ROUTINE
44 *	DO:	KEPLER_ROUTINE
45 *	X = SIGN(DELT) XMAX / 2;	KEPLER_ROUTINE
46 *	DELT_CPRIME, X_CPRIME = 0;	KEPLER_ROUTINE
47 *	END:	KEPLER_ROUTINE
48 *	IF DELT >= 0 THEN	KEPLER_ROUTINE
48 *	XMIN = 0;	KEPLER_ROUTINE
49 *	ELSE	KEPLER_ROUTINE
49 *	DO:	KEPLER_ROUTINE
50 *	XMIN = -XMAX;	KEPLER_ROUTINE
51 *	XMAX = 0;	KEPLER_ROUTINE
52 *	END:	KEPLER_ROUTINE
53 *	DELT = X - X_CPRIME;	KEPLER_ROUTINE
54 *	LOOPING = ON;	KEPLER_ROUTINE
55 *	DO FOR I = 0 TO (I_MAX - 1) WHILE LOOPING;	KEPLER_ROUTINE
56 *	XI = ALPHA X ² ;	KEPLER_ROUTINE
57 *	CALL UNIVERSAL_KEPLER(C1, C2, X, XI, ROMAG) ASSIGN (DELT_C, S_OP_XI, C_OP_XI);	KEPLER_ROUTINE
58 *	TERR = DELT - DELT_C;	KEPLER_ROUTINE
59 *	IF ABS(TERR) >= ABS(EPS_T DELT) THEN	KEPLER_ROUTINE
59 *	DO:	KEPLER_ROUTINE
60 *	CALL SECANT_ITER(DELT_C, DELT_CPRIME, T_FRP, Y) ASSIGN(XMIN, XMAX, DELX, S);	KEPLER_ROUTINE
61 *	IF ABS(DELX) >= EPS_X THEN	KEPLER_ROUTINE
61 *	DO:	KEPLER_ROUTINE
62 *	DELT_CPRIME = DELT_C;	KEPLER_ROUTINE
63 *	X = X + DELX;	KEPLER_ROUTINE
64 *	END:	KEPLER_ROUTINE

SYMT	SOURCE	CURRENT SCOPE
65 *1	ELSE	KEPLER_ROUTINE
E1		
65 *1	LOOPING = OFF;	KEPLER_ROUTINE
66 *1	END;	KEPLER_ROUTINE
67 *1	ELSE	KEPLER_ROUTINE
E1		
67 *1	LOOPING = OFF;	KEPLER_ROUTINE
68 *1	END;	KEPLER_ROUTINE
E1		
69 *1	CALL EXTRAP_STATE(P0, V0, X, XI, S_OF_XI, C_OF_XI, DELT_C) ASSIGN(P, V);	KEPLER_ROUTINE
70 *1	X_C = X;	KEPLER_ROUTINE
71 *1	RETURN;	KEPLER_ROUTINE
72 *1	CLOSE KEPLER_ROUTINE;	KEPLER_ROUTINE

B L O C K S U M M A R Y

PROCEDURES CALLED:
 UNIVERSAL_KEPLER, SECANT_ITER, EXTRAP_STATE

OUTER VARIABLES REFERENCED:
 P, V

EXPLICITLY DECLARED VARIABLES:
 X_C, DELT_C, ROMAG, C1, C2, ALPHA, XMAX, P, XMIN, DELX, I, XI, S_OF_XI, C_OF_XI, TERR, T_EPR, S

The Workshop Philosophy

Each afternoon, I will preside over a HAL/S "workshop" session which will be of interest to those technically-oriented individuals attending the course. The primary purpose of these sessions is to supplement the lectures in several areas:

1. In the first portion of each afternoon, I will field written and/or oral questions upon the lecture material or any related HAL/S topic.
2. On Monday, I will present a HAL/S application problem to be designed and coded by each student. The particular application coded can be either the one I suggest, or a program with which the individual student is familiar. The major portion of each workshop will be devoted to independent work by each student on his problem.
3. For those who are interested, I will be available for informal discussions of HAL/S, the compiler system, run-time characteristics, and related topics during the workshop periods.

Due to the time schedule of this course, the workshops on Monday and Tuesday are pre-mature in relation to the lecture presentation material. To remedy this, and also to serve as a guide to the lectures, I am handing out a document entitled, "HAL/S Language Forms" which outlines the course material and contains sufficient syntactic and semantic information for each student to begin designing his application programs.

In order to make the workshop portions interesting and informative, for all concerned, I would like each student to prepare the following:

1. During each lecture, a list of questions, so that all ground will be covered thoroughly during the question period. These may be given to me just prior to the lunch break, or held until the afternoon session.
2. During Monday's lecture, think about a HAL/S application with which he is familiar - to code as a HAL program during the Monday and Tuesday workshops.