This report describes a programming language for the PDP-11/45 intended for writing operating systems in particular. The language is process, type-less, and priority-less.

It's syntax is similar to that of other high level languages, such as PASCAL and KLISP, but most machine facilities are accessible, and assembly language statements may be freely intermixed with the high level ones.

Special attention has been given to good debugging facilities.

Andrew S. Tanenbaum

Informatica Rapport IR-3
ABSTRACT

This report describes a programming language for the PDP-11/45 intended for system programming in general, and writing operating systems in particular. The language is goto-less, type-less, and priority-less. It's syntax is similar to that of other high level languages, such as PASCAL and BLISS, but most machine facilities are accessible, and assembly language statements may be freely intermixed with the high level ones. Special attention has been given to good debugging facilities.
ACKNOWLEDGMENTS

I would like to thank Ed Keizer and Sape Mullender for help both in designing the language, and in implementing the translator.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Structure of a SAL program</td>
<td>3</td>
</tr>
<tr>
<td>3.</td>
<td>Procedures</td>
<td>4</td>
</tr>
<tr>
<td>4.</td>
<td>Data</td>
<td>6</td>
</tr>
<tr>
<td>5.</td>
<td>Declarations</td>
<td>7</td>
</tr>
<tr>
<td>5.1</td>
<td>LOCAL declarations</td>
<td>7</td>
</tr>
<tr>
<td>5.2</td>
<td>EXTERNAL declarations</td>
<td>8</td>
</tr>
<tr>
<td>5.3</td>
<td>TABLE declarations</td>
<td>11</td>
</tr>
<tr>
<td>5.4</td>
<td>FIELD declarations</td>
<td>14</td>
</tr>
<tr>
<td>5.5</td>
<td>EQUATE declarations</td>
<td>15</td>
</tr>
<tr>
<td>6.</td>
<td>Expressions</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>Operands</td>
<td>17</td>
</tr>
<tr>
<td>6.2</td>
<td>Operators</td>
<td>18</td>
</tr>
<tr>
<td>6.3</td>
<td>Arithmetic expressions</td>
<td>19</td>
</tr>
<tr>
<td>6.4</td>
<td>Boolean relations and expressions</td>
<td>20</td>
</tr>
<tr>
<td>6.5</td>
<td>Simple expressions</td>
<td>22</td>
</tr>
<tr>
<td>7.</td>
<td>Executable statements</td>
<td>24</td>
</tr>
<tr>
<td>7.1</td>
<td>LET statement</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>IF statement</td>
<td>25</td>
</tr>
<tr>
<td>7.3</td>
<td>WHILE statement</td>
<td>26</td>
</tr>
<tr>
<td>7.4</td>
<td>REPEAT statement</td>
<td>27</td>
</tr>
<tr>
<td>7.5</td>
<td>DO FOREVER statement</td>
<td>27</td>
</tr>
<tr>
<td>7.6</td>
<td>FOR statement</td>
<td>28</td>
</tr>
<tr>
<td>7.7</td>
<td>CASE statement</td>
<td>29</td>
</tr>
<tr>
<td>7.8</td>
<td>PRINT statement</td>
<td>31</td>
</tr>
<tr>
<td>7.9</td>
<td>CALL statement</td>
<td>32</td>
</tr>
<tr>
<td>7.10</td>
<td>RETURN statement</td>
<td>34</td>
</tr>
<tr>
<td>7.11</td>
<td>EXITLOOP statement</td>
<td>35</td>
</tr>
</tbody>
</table>
8. Data segments
9. Comments
10. Optimization
11. Debugging and performance monitoring
   11.1 Assertions
   11.2 Flow tracing
   11.3 Variable tracing
   11.4 Single stepping
   11.5 Spider
   11.6 DEBUG statement
   11.7 Subscript checking
   11.8 Procedure call monitoring
   11.9 Program statistics
   11.10 Enabling and disabling individual debugging features
12. Assembly code
13. Built-in functions
14. Idiosyncracies
15. Operating instructions
16. Error messages
17. Context free grammar for SAL
   17.1 Program structure
   17.2 Procedure structure
   17.3 Declarations
   17.4 Procedure bodies
   17.5 Control statements
   17.6 One liners
   17.7 Expressions
   17.8 Data segments
   17.9 Miscellaneous
   17.10 Options
Appendix
1. Introduction

This report is the programmer's manual for a system programming language for the PDP-11, called SAL. This language bears some similarity to other implementation languages such as PL/360 and BLISS. Like these languages, it has the syntax and appearance of a high level language, but the access to machine features usually found only in assembly language. It is intended for writing general system software, such as operating systems, compilers, linkers etc.

Some of major features of SAL are summarized below.

Simple Statements

1. LET statement (assignment)
2. PRINT statement
3. CALL statement
4. RETURN statement
5. EXITLOOP statement

Control Statement

6. IF statement
7. WHILE statement
8. REPEAT statement
9. CASE statement
10. FOR statement
11. DO FOREVER statement

Features aiding debugging

12. Single source statement execution mode
13. No GO TO Statement
14. Assertion checking
15. Variable tracing
16. Procedure call and return tracing
17. Loop tracing
18. Symbolic debugging program
19. Optional run time subscript checking

Data
20. No data typing
21. One dimensional arrays
22. Character arrays
23. Mechanism for accessing bit fields within a word
24. Mechanism for accessing fields of a table
25. Use of both decimal and octal numbers allowed

Other features
26. Recursive procedures allowed
27. Function calls
28. Independent compilation of procedures
29. Direct access to PDP-11's registers
30. Local variable names may be arbitrarily long
31. Designed for ease of typing, e.g. IF and not 'IF'
32. Efficient procedure call mechanism (call by reference)
33. Parameterless macro facility

It is suggested that the reader briefly examine the sample programs in the appendix before reading further, to get an idea of what SAL programs look like.
2. Structure of a SAL program

A SAL program consists of one or more independently translated modules. A module is either a procedure or a data module. Variables declared within a procedure are local to that procedure and cannot be accessed outside of it, except when passed as actual parameters. Variables declared in data modules may be directly accessed in any or all procedures. Thus data modules are used to share data efficiently between many procedures.

A SAL procedure consists of a series of declarations and statements. The 6 control statements IF, WHILE, REPEAT, CASE, FOR, and DO FOREVER may span several lines, but the other statements and declarations must be contained on a single line. This eliminates the need for annoying semicolons between statements. A line beginning with \( \_ \) is deemed to be part of the preceding line. A line may be broken up this way only at a place where a space is allowed. In other words, \( \_ \) begins continuation lines.

Certain words are reserved as key words, delimiters etc. and should not be chosen as identifier names. The reserved words are:

ASSERTION, BY, BYTE, CALL, CASE, DATA, DO, DOWNTO, ELSE, END, EQUATE, ESAC, FI, FIELD, FOR, FOREVER, FROM, IF, IN, LET, LITNU, LOCAL, OD, OUT, PC, PRINT, PROC, REPEAT, RETURN, RO, R1, R2, R3, R4, R5, SP, STATIC, TABLE, THEN, TO, TRACE, UNTIL, UNTRACE, WHILE, TRUE, FALSE, EXITLOOP, BITS

Furthermore, no spaces may be put "inside" these words and no spaces may be put "inside" identifiers. As a consequence, no stropping is needed, i.e. one writes IF, WHILE, END, etc., instead of 'IF', 'WHILE', 'END' etc. as in ALGOL. Experience shows this to greatly speed up typing of programs.
3. Procedures

A procedure begins with a PROC statement (possible preceded by an OPTIMIZE statement and some options or EQUATE declarations) and ends with an END STATEMENT, for example:

```
PROC SUM (A,B,C)
LET A = B + C
END
```

The word PROC is followed by the procedure name.

**IMPORTANT:** procedure names are handled by the PDP-11 DOS linker and therefore must conform to DOS rules. Procedures names are 1-6 letters and digits, beginning with a letter. Underlines are not allowed in procedure names, although they are allowed in local variable names. Furthermore, no two procedure names may start with the same 4 characters.

The formal parameter list, if any, follows the procedure name. Procedures need not have parameters. Formal parameter names follow the same rules as local variables, i.e. they may be arbitrarily long and may contain underlines.

If a formal parameter is an array, its bounds must be indicated in square brackets following the parameter. If an array is a byte (character) array, the word BYTE must follow the bounds. The bounds must be integers. Variable bounds are not permitted. If a procedure will be called with arrays of different sizes as parameters, their lower bounds must all be the same, and the formal bound must be as large as the upper bound of the largest array expected. Bounds may be negative, zero or positive.

If the proc name also appears after END, that proc will be taken as the starting address by the linker.
Some examples of PROC statements follow:

PROC NULLE
PROC INCR (X)
PROC VECADD (A [0:100], B [0:100], C [0:100])
PROC HORSE (HIND_LEG, NAME [1:10] BYTE, SIZE)
SAL is an untyped language. This means that the language does not make any distinction between integers, booleans and other data types. Furthermore, any operation may be performed on any data object.

A data object is characterized by a single parameter: the number of bits it contains. A scalar variable or a register contains 16 bits, i.e. one PDP-11 word. One dimensional arrays of 16 bit words are also provided, as are arrays of 8 bit bytes. It is also possible to handle data in units of 1 to 16 bits by using fields and selections, as described in section 5.

Data objects smaller than 16 bits can be regarded as unsigned positive integers. Thus, if a 4 bit field containing 1111 is extracted from a word and assigned to a 16 bit scalar variable, the variable will acquire the value 15, not -1. Similarly, an element of a byte array may contain an integer between 0 and 255, inclusive, but not a negative integer. In other words, when a data object of less than 16 bits is converted to a 16 bit quantity, it is right justified with zero fill on the left end.

Local variable names are "identifiers". This means they may be arbitrarily long and may contain any combination of upper case letters, digits and underlines, the first of which must be an upper case letter.

Examples of identifiers:

A, X12, Q44, CONF_NUMBERS, A_B_C

Spaces are not permitted within identifiers.

A local declaration may declare scalar variables (16 bit words), word arrays and byte arrays. Scalar-byte variables do not exist. An array is declared by following the identifier with the bounds, in square brackets. A byte array has the word BYTE after the closing bracket. The bounds must be
5. Declarations

A procedure may contain certain variable and other declarations. All declarations must come at the beginning of the procedure, before any executable statements. The order of the declarations is not important.

There are 5 declarations:

1. LOCAL declarations
2. EXTERNAL declarations
3. TABLE declarations
4. FIELD declarations
5. EQUATE declarations

5.1 LOCAL declarations

LOCAL declarations are used to declare local scalar (i.e. 1 word) variables, and local arrays. When a procedure is entered a block of storage is reserved on the stack for the local variables. After the procedure has been left, the local storage is released.

Local variable names are "identifiers". This means they may be arbitrarily long and may contain any combination of upper case letters, digits and underlines, the first of which must be an upper case letter.

Examples of identifiers:

A, XYZ, Q44, GOAT_NUMBER, A_B_C

Spaces are not permitted within identifiers.

A local declaration may declare scalar variables (16 bit words), word arrays and byte arrays. Scalar byte variables do not exist. An array is declared by following the identifier with the bounds, in square brackets. A byte array has the word BYTE after the closing bracket. The bounds must be
negative, zero, or positive integers and no bound may be less than -16384
or greater than 16383.

Examples of LOCAL declarations:

    LOCAL I, J, K
    LOCAL A, B, C [-10:10], D [10:20], E [0:5] BYTE, F
    LOCAL OH_MY_GOODNESS_I_AM_LATE_AGAIN

The above 3 statements declare 7 scalars, 2 word arrays, and 1 byte array.
Each array variable must have its own bounds following it. Otherwise ambiguities
could result.

5.2 EXTERNAL declarations

EXTERNAL declarations are used to access variables declared in data
modules. The format is the same as for LOCAL declarations.

Examples:

    EXTERNAL ABC, XYZ, HORSE [1:10], P
    EXTERNAL HAVE_A_NICE_DAY, WW [3:5], LINE [0:80] BYTE, COUNT

No stack storage is reserved for external variables. Instead storage is reserved
within the data module in which they are declared (by a STATIC declaration).
External variables are similar to variables declared in the outer block of an
ALGOL program, i.e. they can be used in all procedures. In ALGOL global
variables need not be declared. As a consequence it is easy for a procedure
with an error in it to cause trouble in other procedures which use global
variables it has accidentally changed. In SAL the changes are reduced, since a
procedure can only access external variables that are explicitly declared.

External variables are linked by the DOS loader, and therefore must be
1-6 characters with no underlines. A single EXTERNAL declaration may declare
variables located in several data modules. An external variable must not have
the same name as a procedure (first 4 characters must be unique).

External variables are also used to access the PDP-11 device registers and interrupt vectors. Each device register and interrupt vector has a predefined name. By declaring these names in a module, the corresponding device register or interrupt vector can be accessed. For example, the RK05 disk word count register (777406) can be set using this procedure:

```
PROC SETWC (WORD_COUNT)
EXTERNAL RKWC
LET RKWC = WORD_COUNT
END
```

If the EXTERNAL declaration above had been omitted, RKWC would have been flagged as an undefined symbol. RKWC (and all other device and vector names) may be used as a local name, provided it is not being used to access the corresponding device or vector, i.e. these names are not "reserved" in the same sense as IF, WHILE, FOR etc. This effect is similar to an ALGOL programmer's ability to use SQRT as a name for one of his procedures, providing he does not need the library SQRT.

The list of named device registers and vectors follows. Note that MEMORY [N] can be used to access location N.
5.3 TABLE declarations

TABLE declarations are used to provide access to external tables.
A table is an array of structures with several components. A table is similar
to an array of records in PASCAL, or a row of STRUCT’s in ALGOL 68.

As an example of a table, consider the following:

<table>
<thead>
<tr>
<th>SEG 1</th>
<th>SEG 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEG 3</td>
<td>SEG 2</td>
</tr>
<tr>
<td>SEG 5</td>
<td>SEG 4</td>
</tr>
<tr>
<td>SEG 7</td>
<td>SEG 6</td>
</tr>
</tbody>
</table>

\[
\text{LENGTH} \quad \text{LENGTH} \\
\text{R} \quad \text{R} \quad \text{E} \quad \text{X} \quad \text{R} \quad \text{E} \quad \text{X} \\
\text{LINK} \quad \text{LINK} \\
\text{T [1]} \quad \text{T [2]} \\
\]

This table, T, might consist of 100 entries, each of which occupies 6 words.
Two entries, are shown above. Each entry is divided into components. A
component may either be a scalar of 1 - 16 bits, or a one dimensional array
of scalars (of any length). Each component has a name, and it is these names
that are used to select components out of tables. For example,

\[
\text{LET LINK OF T [4] = 0} \\
\]

assigns the LINK component of T [4] the value 0. Similarly,
IF SEG [K] OF T [N+1] = 0 THEN....

tests the K-th element of the SEG array (0 ≤ K ≤ 7) in entry N + 1 of T.

A component name (optionally subscripted) followed by OF and a table entry is called a selection. The component name is called the selector. A table may have arbitrarily many entries, and each entry may have an arbitrary number of components. Tables allow data to be packed to economize on storage.

A TABLE declaration simply lists the tables used. For example,

TABLE CBT, FHT, HNT, ABC

The names must conform to DOS naming rules, i.e. 1 - 6 characters with no underlines.

When a TABLE declaration is encountered the translator generates a MACRO-11 command to fetch a macro definition from the system macro file. The macro name is the table name prefixed by $. Thus:

TABLE MILLION yields .MCALL $MILLION.

The user must write, and, if authorized, install his table macros himself.

When a TABLE is used in a SAL program, the translator generates a macro call of the form

$MILLION SELCTR, OP

Where MILLION represents the table name, SELCTR represents the selector (as a character string) and OP is an integer from 0 to 8 inclusive specifying the operation to be performed. The entry number will be in R3 and the selector subscript, if there is one, will be on top of the stack. The macro may change R3 and R4 as well as the top 2 words of the stack, but must restore any other
registers modified, except R5 as described below. SP must not be changed.

The 9 operations are as follows.

0 - load the component into R5
1 - add the component to R5
2 - subtract the component from R5
3 - multiply the component into R5
4 - divide R5 by the component
5 - boolean AND the component into R5
6 - boolean OR the component into R5
7 - boolean XOR the component into R5
8 - Store R5 into the component.

When a component shorter than 16 bits is used, it must first be converted
to a 16 bit number, right justified with zero fill at the left end. Because
the user may write his own macros, he may use any packing format and any access
algorithm he needs. A typical macro will look like this

.MACRO $MILLION, SELCTR, OP
  .IF IDN, <SELCTR>, <NAME1>
      : (fetch NAME1)
  .MEXIT
  .ENDC
  .IF IDN, <SELCTR>, <NAME2>
      : (fetch NAME2)
  .MEXIT
  .ENDC
  :
  .ENDM $MILLION
This method of structure definition is somewhat unusual, but it has several advantages over the more conventional definition facilities. First, and most important, because the macros must be preloaded on the file SALMAC. SML (by the project manager) it is automatically guaranteed that all the people working on a large project will, in fact, actually be using precisely the same data structures. If each person has to provide his own specification, there is a very serious problem maintaining consistency when the data structures are changed, which happens fairly often in the course of developing a large system. Some people always forget to update all their programs when a memo arrives announcing yet another change in some data structure.

Second, the programmer can utilize any sequence of machine instructions in accessing the data, providing maximum flexibility.

5.4 FIELD declaration

FIELD declarations are used to access fields within a word. This allows bits to be packed and unpacked easily.

Examples:

FIELD LOBYTE = BITS 0 TO 7
FIELD XYZ = BITS 4 TO 12
FIELD SIGN = BITS 15 TO 15

Fields are used to extract or store partial words. For example:

LET BOAT = GOAT. XYZ

will extract bits 4 - 12 of GOAT, right shift them into bits 0 - 8 of BOAT and set bits 9 - 15 of BOAT to zero. Furthermore

IF X.LOBYTE = 0 THEN...
will test the low order byte of the variable X to see if it is zero.

In essence, a field is a mask which can be placed "on top of" any scalar variable to extract or store bit fields without affecting the other bits.

Each field declaration occupies an entire line. The bits are numbered with bit 0 being the low order (rightmost) bit, and bit 15 being the sign (leftmost) bit.

5.5 EQUATE declarations

The EQUATE statement provides the programmer with a method for giving a symbolic name to any piece of text. When the symbolic name is encountered in the program, it is systematically replaced by its defined text. Thus EQUATE statements can be used as a simple parameterless macro facility. The replacement occurs before the statement in which it appears is syntactically analyzed.

Two examples of the uses to which EQUATE can be put follow. One use is to allow constants to be given symbolic names. For example

```
PROC BEAST
EQUATE ARRAY_SIZE = 100
LOCAL A [0:ARRAY_SIZE], B[0:ARRAY_SIZE], C[0:ARRAY_SIZE]
```

makes it possible to change the size of all 3 arrays by only changing one line. Because it is generally considered poor programming practice to have unidentified constants in a program, cf.

```
LET N = N - 347

with
EQUATE HASH_TABLE_SIZE = 347
LET N = N - HASH_TABLE_SIZE
```
EQUATE statements can help the readability of programs. Because the symbolic name is replaced by the constant itself before the statement it appears in is parsed, the generated machine code can take advantage of efficiencies not possible with a variable. For example, consider the following 2 program fragments:

PROC BAD
LOCAL N,A [1:20]
LET N=10
LET A [N] = 4

PROC GOOD
LOCAL A [1:20]
EQUATE N=10
LET A [N] = 4

In the latter procedure, the parser "sees"

LET A [10] = 4

because the replacement of N by 10 has already occurred. Since the address of A [10] is known at compile time, a single MOV instruction can be generated, whereas in the former case it is much harder to generate optimal code without doing extensive flow analysis of the entire program.

A second use for EQUATE declarations is to assign a new symbolic name to a machine register (or device register or interrupt vector). For example

EQUATE COUNTER = RO

This allows the name COUNTER to be used as a variable, and to have it stored in RO, instead of in memory.

The general format for an EQUATE declaration is a single line beginning with EQUATE, followed by an identifier, an equals sign, and any text.
6. Expressions

Expressions are used in many places in SAL, among them, the condition in an IF or WHILE Statement, and the right hand side of an assignment statement. An expression is a rule for computing a value, a 16 bit number, that can then be tested or moved somewhere.

6.1 Operands

Expressions are composed of alternating series of operators and operands. There are 10 types of operands:

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Scalar variables</td>
<td>X, N, CUE_BALL</td>
</tr>
<tr>
<td>2. Array elements</td>
<td>A [I], SLOT [2*K+1]</td>
</tr>
<tr>
<td>3. Decimal integers</td>
<td>0, 999, 22</td>
</tr>
<tr>
<td>4. Octal integers</td>
<td>OCTAL (177200), OCTAL (0006)</td>
</tr>
<tr>
<td>5. Registers</td>
<td>RO, R1, R2</td>
</tr>
<tr>
<td>6. Function calls</td>
<td>LENGTH (N), F(X), READ ()</td>
</tr>
<tr>
<td>7. Scalar selections</td>
<td>TAIL OF DOG [N], KNEE OF NEWT [1]</td>
</tr>
<tr>
<td>9. Bit fields</td>
<td>PRS. BUSY_BIT</td>
</tr>
<tr>
<td>10. Asc pseudofunction</td>
<td>ASC (n)</td>
</tr>
</tbody>
</table>

A scalar variable or array name may be local, external or a parameter. External variables are restricted to a maximum length of 6 characters, but local names and parameters may be arbitrarily long and may contain underlines.

Decimal integers must not contain or be followed by a decimal point, and must lie in the range 0 to 32767.

Octal integers must be in the range 0 to 177777.
The PDP-11 registers R0, R1, and R2 are available for general use by the programmer, however the other 5 registers are used by the translator for various purposes and should not be used. Furthermore, assembly language programs called from SAL programs should be sure to restore the registers other than R0-R2 to their original values, upon returning. R0-R2 are not automatically saved on a procedure call. In effect, the translator does not know that they even exist.

Function calls consist of a name and a (possible empty) parameter list. Passing of function names as parameters is permitted. Function names so passed must be declared in EXTERNAL declarations. If a function has no parameters, () must follow its name to indicate a function call.

LET N = READ()

The ASC pseudofunction takes a single character as argument, and returns the ASCII code for it. Only upper case characters are allowed. E.g.

IF N = ASC (K) THEN CALL KILL FI

6.2 Operators

Fifteen operators are provided to combine operands:

Operator | Meaning
--- | ---
1. `+` | Addition
2. `-` | Subtraction
3. `*` | Multiplication
4. `/` | Division
5. `&` | Boolean AND of 16 bit numbers
6. `!` | Boolean OR of 16 bit numbers
7. `#` | Boolean EXCLUSIVE OR of 16 bit numbers
8. `<` | Less than
9. `<=` | Less than or equal to
10. =
11. /=
12. >
13. >=
14. AND
15. OR

Equal to
Not equal to
Greater than
Greater than or equal to
Used to combine required relations
Used to combine alternate relations

Overflow is not detected on any operation

6.3 Arithmetic expressions

An arithmetic expression is a strictly alternating sequence of operands and the operators + - * / & ! #. An arithmetic expression may begin with a leading + or - sign. Unlike ALGOL, but like APL, SAL does not have operator priorities. In SAL, all arithmetic expressions are evaluated strictly left to right. This produces very efficient code. As an example, consider the arithmetic expression A * B + C * D.

The equivalent ALGOL expression is ((A*B)+C)*D.

The code generation algorithm in SAL uses R5 as an accumulator. A * B + C * D is translated to

```
MOV A, R5
MUL B, R5
ADD C, R5
MUL D, R5
```

Operator precedence implies the use of temporary variables, which in turn implies complicated registers optimization algorithms etc, and is in direct opposition to the whole philosophy of SAL as an efficient replacement for assembly language. If a programmer wants to evaluate an expression involving temporaries, he may use R0 - R2 explicitly. For example, the ALGOL assignments
X : = A * B + C * D can be evaluated by
LET RO = A * B
LET X = C * D + RO

Note that parentheses are not allowed in arithmetic expressions, except to enclose function parameters. Thus
LET X = (A * B) + (C * D)
is a syntactic error.

Quantities shorter than 16 bits are extended to 16 bits by right justifying them with zero fill before being used in arithmetic expressions.

Some examples of arithmetic expressions:

- X
X.LOBYTE * RO/4 - A [R1]
N & OCTAL (177) ! 1
KNEE OF NEWT [3*K+1] - DEVICE.STATUS
Z - F ()
- X + 3 - B [R1+2]

6.4 Boolean relations and expressions

A boolean relation consists of 2 arithmetic expressions separated by one of the relational operators.
<, <=, =, /=, >, or >= or an arithmetic expression (see below). A boolean relation is true if the specified condition is met, and false otherwise.

Comparisons are made regarding both operands as signed, two's complement 16 bit numbers. Thus machine addresses should not be compared this way, since addresses above 32767 will appear negative.

An arithmetic expression can also be used as a boolean relation, with 0 meaning false, and everything else meaning true.

A boolean expression is either a boolean relation, or several boolean relation joined together by the connectives AND and OR. If a boolean expression contains AND's but no OR's, the value is true if and only if all the relations
are true. If a boolean expression contains OR's but no AND's, the result is true if any of the relations are true. There is no guarantee that an entire boolean expression will be evaluated if the result can be determined without doing so. For example, in

\[ 1 > 2 \text{ AND } P(x) = 0 \]

The function may or may not be called depending on the whims of the implementer. This is to discourage functions with side effects.

When AND's and OR's are combined in a single boolean expression, the expression is evaluated left to right. Consider the following boolean expression:

\[ P \text{ AND other-boolean-expression.} \]

where \( P \) represents a boolean relation. If \( P \) is false (i.e. 0) the whole thing is false. If \( P \) is true (i.e. \( \neq 0 \)) the result of the whole thing is the result of the other-boolean-expression. Similarly, if the left part of an OR is true, the whole thing is true, and if the left part is false, the result of the whole thing is the result of the right part. In this way evaluation proceeds from left to right. Parentheses may not be used to change the order of evaluation. Thus

\[ K = P(x) \text{ AND } M = D(x) \]

\[ P \text{ AND } Q \text{ OR } R \text{ means } P \land (Q \lor R) \]

\[ P \text{ OR } Q \text{ AND } R \text{ means } P \lor (Q \land R) \]

**TRUE** and **FALSE** are reserved words defined as 1 and 0 respectively.

A boolean expression may be used on the right hand side of an assignment statement. If the result is true, the value 1 is assigned, otherwise the value 0 is assigned.

For example,

\[ \text{LET } N = A < B \text{ AND } C = D + 1 \]
will assign N either 0 or 1 depending upon A, B, C, and D.

Note that AND has nothing to do with the operator &, and OR has nothing to do with the operator !. Furthermore both of the following are valid, although different.

\[
\text{LET } M = A \& B
\]

\[
\text{LET } N = A \text{ AND } B
\]

In the first case the 16 bit boolean product of A and B will be formed and stored in M. In the second case N will be assigned the value 1 if neither A nor B is 0, and N will be assigned 0 if either A or B or both is 0. Thus if \( A = 26 \) and \( B = 12 \), we get \( M = 16 \) and \( N = 1 \). The latter statement really means

\[
\text{LET } N = A \neq 0 \text{ AND } B \neq 0
\]

Some examples of boolean expressions are:

\[
\text{THE\_THREE\_LITTLE\_PIGS}
\]

\[
A < B
\]

\[
\text{T A I L \ OF \ D O G [N+1]} > \text{T A I L \ OF \ D O G [N+2]}
\]

\[
K = 0 \text{ AND } N = 0 \text{ AND } L = 0
\]

\[
K = F(X) \text{ AND } M = G(X)
\]

\[
A [1] \leq A [I+1] \text{ OR } \text{FLAG} = \text{FALSE}
\]

\[
\text{TCCM.\_ERROR\_BIT} = 1 \text{ OR } \text{TCCM.\_READY\_BIT} = 1
\]

TRUE and FALSE are reserved words defined as 1 and 0 respectively.

6.5 Simple expressions

Arbitrary (boolean) expressions may be used in most places in SAL, however, a more restricted class of expressions is also used in certain syntactic positions, namely in subscripts (including in selections) and as actual parameters in a function or procedure call. These are called simple expressions.
A simple expression may involve only 3 kinds of operands:

1. scalar variables
2. Registers
3. Decimal integer constants.

Names EQUATED to any of the above may also be used, of course, as well.

Furthermore, only the following 3 operators may be used:

1. +
2. -
3. *

Simple expressions are also evaluated left to right. Examples of simple expressions:

```
X
X + R0
3 * N + 2
- 6 - K
```
7. Executable Statements

In this section the SAL statements will be covered.

7.1 LET statement

The LET statement is used to assign values. The right hand side of a LET statement may be any expression. The left hand side must be one of the following (or a name EQUATED to one of the following):

1. Scalar variable
2. Array element
3. Register
4. Selection
5. Bit field

If a LET statement has a 16 bit quantity on the left hand side and a shorter quantity on the right hand side (e.g. bit field, selection from a table, or an element of a byte array) the right hand side is first right justified in a 16 bit internal variable with zero fill at the left, and then assigned. When the left hand side is a bit field or a selection, only the specified field or component is modified; the other ones are not affected. For example, in

FIELD MIDDLE = BITS 4 TO 11
LET N.MIDDLE = K

only bits 4-11 of N are changed. Bits 0-3 and 12-15 are not affected.

If the right hand side is a 16 bit quantity but the left hand side is shorter, only the low order bits of the right hand side are used.

If neither side is 16 bits, the right hand side is first extended to 16 bits (in a temporary) by right justifying it and clearing the unused
high order bits to 0. Then the appropriate number of low order bits are
used in the assignment, where the left hand side determines how many bits
are appropriate.

A LET statement must be contained on 1 line (which may be continued
by the\texttt{\textbackslash})\texttt{\textbackslash}). Some examples are:

\begin{verbatim}
LET COUNT = 0
LET TCCM_FUNCTION = 2
LET PRFSW = OCTAL (320)
LET TAIL OF DOG [3*N+1] = A < B + 1
\end{verbatim}

\textbf{7.2 IF statement}

The IF statement takes one of 2 forms:

1. \texttt{IF \langle expression\rangle THEN \langle group\rangle FI}

2. \texttt{IF \langle expression\rangle THEN \langle group\rangle ELSE \langle group\rangle FI}

In both cases an expression may be an arithmetic expression (in which case 0
means false, non zero means true). The IF statement is terminated by \texttt{FI}. An
IF statement may extend over arbitrarily many lines, but the expression must
be contained on 1 line.

Between the \texttt{THEN} and \texttt{FI}, \texttt{THEN} and \texttt{ELSE} and \texttt{ELSE} and \texttt{FI} arbitrarily
many statements may appear. \texttt{BEGIN} and \texttt{END} are neither required nor permitted.
Any executable statement may follow \texttt{THEN} or \texttt{ELSE}. 
Examples of IF statements:

IF X < Y THEN LET R2 = 0 ELSE LET R2 = 1 FI

IF X < Y AND Z = 0

THEN LET Z [4] = R0 + R1


ELSE IF W = 0

THEN CALL SUB1 (A, B)

PRINT A, B

LET K = K + 1

ELSE PRINT "ERROR 6"

FI

FI

IF W + 1 > = 0 THEN CALL LOAD FI

7.3 WHILE statement.

The WHILE statement has the form

WHILE <expression> DO <group> OD

The expression follows the same rules as IF. Similarly any statement or statements may occur between DO and OD. An empty group is also allowed.

Examples of WHILE statements:

WHILE X = 0 DO CALL SUB (X) OD

WHILE I < N

DO IF A [I] = 0

THEN EXITLOOP

ELSE LET A [I] = - A [I]

FI

OD
7.4 REPEAT statement

The REPEAT statement has the form

REPEAT <group>
UNTIL <expression> LITNU

The REPEAT statement is similar to the WHILE statement, one difference being that in REPEAT the test is made at the end of the loop rather than at the beginning. This means that the statements between REPEAT and UNTIL will always be executed at least once. The UNTIL and LITNU (UNTIL backwards) must be on the same line.

7.5 DO FOREVER statement

The DO FOREVER statement has the form

DO FOREVER <group> OD

This statement is equivalent to

WHILE 1 + 1 = 2 DO <group> OD.

DO FOREVER is more efficient, because no test is performed, and furthermore the fact that the loop is non-terminating is clearer to the reader.

A example of the use of DO FOREVER is as follows. Operating systems typically have modules that get messages, handle these messages, and then repeat. This can be programmed as:

PROC BEAST
LOCAL A [1:100]
DO FOREVER
  CALL GET (A)
  CALL HANDLE (A)
OD
Examples of FOR statements.

FOR I FROM 1 TO 10 DO LET A[I] = 0 OD
FOR I FROM N + K TO Q * Q BY K - 3 DO LET A[I] = I OD
FOR I FROM 10 DOWNTO 0 BY - 2 DO LET A[I] = 0 OD
FOR I FROM 20 DOWNTO - 20 DO LET A[I] = 0 OD
FOR EXT FROM F(X) TO G(X) + TAIL OF DOG [4] DO LET A[I] = 0 OD

FOR I FROM 1 TO N BY 2 DO
FOR J FROM 1 TO M BY 3 DO
CALL SUB (I, J)
OD OD

Note that the expressions following TO and BY are evaluated on every iteration through the loop, so it is desirable to make them as simple as possible.

7.7 CASE statement

The CASE statement is a generalization of the IF statement. The IF statement allows a 2 way branch. The CASE statement allows an N way branch.

The CASE statement has the format:

CASE <expression> IN

1 = (<group>),
2 = (<group>),
   ...
<integer> = (<group>),
OUT = (<group>) + optional

ESAC

The numbers in front of each group must be consecutive, beginning at any
integer. The expression is evaluated, and if equal to the number of a clause, the specified clause is executed. If the expression is not equal to the number of a clause, the OUT clause is executed, if present. The OUT clause is optional, and if not present, the CASE statement may be skipped. After a clause has been executed, execution continues with the statement following the ESAC, i.e. at most 1 clause will be executed.

A schematic comparison of IF and CASE may be helpful:

```
 TRUE          FALSE
 IF test       CASE
 THEN PART     1 2 ... N OUT
 ELSE PART
```

Due to a peculiarity of the translator, the numbers in front of the clauses, the equal signs and the left parentheses must be on the same line. Likewise the closing parentheses and following commas must be on the same line. ESAC must appear alone on a line.

Examples of CASE statements:

```
CASE I IN
    1 = (LET X = 4),
    2 = (CALL SUB(X, Y, Z)
       LET P = 0
       IF T = 3 THEN PRINT Y FI),
    3 = (RETURN)
ESAC
```
CASE I + FUN (Z) IN

0 = (RETURN),

1 = (WHILE Z > 0

DO LET A[Z] = 0

LET Z = Z - 1

PRINT

OD),

2 = (CALL SUB 1),

3 = (CALL SUB 2(Z)),

OUT = (PRINT "CASE ERROR", I, Z)

ESAC

CASE 3 * K - N.FOO IN

8 = (CALL A),

9 = (CALL B),

ESAC

7.8 PRINT statement

The PRINT statement is used to produce output of numbers and strings.

The format of a PRINT statement is

PRINT item 1, item 2, ..., item n

An item can be one of 3 possibilities

1. An expression e.g. X + TAIL OF DOG [N]

2. A literal string e.g. "HELLO"

3. An array name e.g. A

Expressions are evaluated and printed as decimal integers in a field 8 spaces wide, right adjusted. This provides room for a leading blank, minus sign,
5 digits, and trailing blank. Literal strings are printed as written, with no leading or trailing blanks. Arrays are assumed to contain characters in ASCII to be printed, one character per byte. (Thus a word array of 10 words will print as much as a byte array of 20 bytes). Printing continues until a zero byte is encountered. The array may be arbitrarily long, and may contain carriage returns and line feeds. Line overflow is not detected.

After the last item has been printed, a carriage return and line feed are printed.

The programmer can control the output device used by PRINT statements by changing the value of the EXTERNAL variable OUTDEV. The allowed values are

0 - DECwriter
1 - GT-40
2 - Paper tape punch
3 - Line printer

Other values are reserved for future use. OUTDEV must be explicitly declared EXTERNAL. OUTDEV may be changed during execution of a program as often as desired.

Examples of PRINT statements

```
PRINT "X =", X, "Y =", Y, "Z =", Z
PRINT A + Z, FUN (Y), "TEST", A + B, 10
```

7.9 CALL statement

CALL statements are used to invoke procedures. There are 2 forms, one with actual parameters, and one without:

1. CALL <procname>

2. CALL <procname> (<parameter>, <parameter>, .... <parameter>)

The parameter mechanism is call by reference. A parameter is passed by putting its address on the stack. The following classes of parameters are
distinguished.

1. Scalar variables (local, parameter or external)
2. Array names
3. Elements of arrays
4. Simple expressions (see section 6.5)
5. Procedure (function) names
6. Literal strings

Formal parameter types must be appropriate to the actual parameter types. Passing elements of byte arrays is allowed, in the sense that the correct calling sequence will be generated, but should not be used when calling SAL programs unless the corresponding formal is a byte array of length 1. Instead, the element could be assigned to a register or dummy variable and that could be passed. If an external procedure name is to be passed, it should be declared EXTERNAL. Literal strings are compiled by .ASC12 rather than .ASCI1 pseudoinstructions, so a zero byte is appended to the end. This allows them to be printed.

As an example of the calling sequence generated, consider the following example.

CALL TEST (LOCL, PARAM, EXT, 44, "ABC")

where LOCL, PARAM and EXT are local, parameter and external scalars (or array names). The generated calling sequence is:

  MOV  #Const1, -(SP) ; LOCL
  ADD  SP, (SP)       ; LOCL
  MOV  Const2 (SP), -(SP) ; PARAM
  MOV  #Ext, -(SP)    ; EXT
  MOV  #44, Const3 (SP) ; 44
MOV #CONST4, -(SP) ; 44
ADD SP, (SP) ; 44
MOV #ADDR, -(SP) ; "ABC"
JSR PC, TEST

where the CONST'S and ADDR are appropriate constants and addresses.

A function may also be called via a CALL statement, in which case the value, returned in R5, is ignored.

Examples of CALL statements:

CALL FOO
CALL PR("HELLO THERE", "CHARLIE BROWN")
CALL HORSE(A,B,BUF[I], 3*K)
CALL GOAT(4)

7.10 RETURN statement.

RETURN statements are used to return from a procedure, i.e. to terminate execution of a procedure and return control to the calling procedure. There are 3 formats:

1. RETURN

2. RETURN <expression>

3. RETURN FROM INTERRUPT

A procedure may return a value, in which case the procedure may be used as a function call or in a CALL statement. To make this clearer, consider the difference between a procedure and an integer procedure in ALGOL 60. The former may not be used in expressions, but the latter may be used wherever an integer is allowed. In ALGOL 60 values are returned by assigning to the procedure name. In SAL values are returned by putting them in RETURN statements. The ALGOL 60 procedure and SAL procedure below are "equivalent".
ALGOL 60: integer procedure sum (a,b); value a, b; integer a, b;
    sum: = a + b;

SAL: PROC SUM (A, B)
    RETURN A + B
END

A SAL procedure may contain several RETURN statements, in which case either all must return expressions or none must return expressions. A RETURN is automatically generated from an END statement, so if a procedure does not return a value it may simply "fall through" and "execute" the END statement. The third form removes the locals and return address, then RTI's. Examples of RETURN statement.

RETURN
RETURN 3 + RO
RETURN N * FACT (N-1)

7.11 EXITLOOP statement.

The EXITLOOP statement is a specialized form of the now-in-disrepute GOTO statement. It exits one level of loop and continues execution with the statement following the loop. It can exit FOR, WHILE, REPEAT, or DO FOREVER loops.

Consider a procedure which takes as its first parameter an array of 100 elements. The procedure searches the array for a value specified by the second parameter. If the array contains an element equal to the second parameter, it must return the index of that element; otherwise it must return 0. The following SAL procedure does this.
PROC SEARCH (A[1 : 100], SEARCH KEY)

LET RO = 0

FOR R1 FROM 1 TO 100

DO IF A[R1] = SEARCH KEY

    THEN LET RO = R1

EXITLOOP

FI

OD

RETURN RO

END

The only example of an EXITLOOP statement is:

EXITLOOP

A data segment consists of a series of declarations of variables and constants that are shared between the procedures without having to be passed as parameters. This kind of information is often called "global variables" as contrasted with "local variables" which are only directly accessible within one procedure. In ALGOL-60, global variables declared in the outermost block of the program are directly accessible to all procedures without any declaration. This has two purposes: first, to allow independently compiled procedures to share data; second, to reduce the chance of a procedure inadvertently changing a global variable.

A data segment consists of the or more procedures and sets of more data segments; a data segment is a module containing only global variable declarations, and no executable statements. Unlike local variables, which are kept on the stack, global variables are static: they occupy fixed locations determined by the loader. Because space for global variables is reserved at load time, and not at procedure entry time, global variables may be initialized. For example, if a large table is needed, it can be loaded along with the procedures. In ALGOL-60 this is not possible. If an expression is written in ALGOL-60 and it needs a table of the symbolic operators (e.g., add, mul, etc.) and their values, it must either read it in or start it off by performing a large number of assignments, both of which waste time and space compared with simply having the loader fill the table when the program is loaded.
8. DATA SEGMENTS.

It is frequently useful to share data between procedures without having to pass the data explicitly as parameters. This kind of information is often called "global variables" as contrasted with "local variables" which are only directly accessible within one procedure. In ALGOL 60 global variables declared in the outermost block of the program are directly accessible to all procedures without any declaration. In SAL an explicit declaration for each global variable is required. This has two purposes: first, to allow independently compiled procedures to share data; second, to reduce the chances of a procedure inadvertently changing a global variable (i.e. to prevent errors).

A SAL program consists of one or more procedures and zero or more data segments. A data segment is a module containing only global variable declarations, and no executable statements. Unlike local variables, which are kept on the stack, global variables are static: they occupy fixed locations determined by the loader. Because space for global variables is reserved at load time, and not at procedure entry time, global variables may be initialized. For example, if a large table is needed, it can be loaded along with the procedures. In ALGOL 60 this is not possible. If an assembler is written in ALGOL 60 and it needs a table of the symbolic opcodes (mov, add, mul, etc) and their values, it must either read it in or start off by performing a large number of assignments, both of which waste time and space compared with simply having the loader fill the table when the program is loaded.

A data segment consists of a

```
DATA SEGMENT n
```

statement where n is 0, 1, 2, 3, 4, 5, 6, or 7, followed by a series of STATIC and EXTERNAL declarations and ended by an END statement. DATA SEGMENT
n can load data into PDP-11 virtual addresses 8192n to 8192n + 8191. The maximum segment size is 8192 bytes.

A STATIC declaration contains a list of scalar variables and word arrays, just as a LOCAL declaration. Array bounds must be integers, and follow the array name, the same as for LOCAL declarations. STATIC variables must have names that conform to DOS standards, i.e. 1 - 6 letters and digits beginning with a letter.

STATIC variables may be initialized by following a variable with its initial value(s) in parentheses. All or part of an array may be initialized by providing a list of values. An initial value must either be a decimal integer, or an external name (e.g. a procedure name) in which case it must be declared in an EXTERNAL declaration within the data segment.

The default initialization is zero.

Examples of STATIC declarations:

STATIC X
STATIC Y, Z(4), N
STATIC A, B[1 : 3] (8, 9, 10), W[4 : 5] (0,4), K
STATIC Q[0 : 3] (PROC1, PROC2, PROC3, PROC4)

An example of global variables in ALGOL 60 and SAL may be helpful.

ALGOL 60 :  begin  integer  i;
    i: = 7;
    procedure  incr;
    i: = i + 1;
    procedure  decr;
    i: = i - 1;
    
end
SAL :
DATA SEGMENT 0
STATIC I (7)
END
PROC INCR
EXTERNAL I
LET I = I + 1
END
PROC DECR
EXTERNAL I
LET I = I - 1
END

Note that variables declared in STATIC declarations are automatically EXTERNAL and need not be declared so.
9. Comments.

A comment is started by // and terminated by a semicolon.

Examples of comments:

// THIS IS A COMMENT;

// THIS IS
ALSO A
COMMENT;

A comment may occur anywhere a space is permitted. Comments are important and programs should contain many of them. A ratio of 1 line of comment to 1 line of code is reasonable.

Two statements are provided for this purpose:

USE BRANCHES
USE JUMPS

These statements cause branches and jumps respectively to be generated until another one of them is encountered. Typically, a USE JUMPS will appear before a large IF or WHILE statement to form use of a jump, and a USE BRANCHES will appear after it to resume generating branches.

These two statements may each appear as often as necessary in a procedure.
10. Optimization.

Two compilation modes are available: debugging and optimization. The default mode is debugging. To use optimization mode, which suppresses various run time checks, the run time symbol table, etc. place an OPTIMIZE statement directly before the PROC statement.

In optimizing mode, the compiler generates branches instead of jumps. For backward branches there is no problem, but for forward branches the compiler cannot tell if a label is within 127 words or not. If the label is too far, an assembly error of type A will occur. In this case it will be necessary to recompile the procedure, instructing the compiler to use jumps where needed.

Two statements are provided for this purpose:

USE BRANCHES

USE JUMPS

These statements cause branches and jumps respectively to be generated until another one of them is encountered. Typically, a USE JUMPS will appear before a large IF or WHILE statement to force use of a jump, and a USE BRANCHES will appear after it, to resume generating branches. These two statements may each appear as often as necessary in a procedure.
11. Debugging and Performance Monitoring

A variety of debugging facilities have been provided to help debug programs. The best strategy is not to make any errors in the first place, but if an occasional error sneaks in, these tools will help you find them quickly.

11.1 Assertions

Assertions are not only useful for proving programs correct, but also can serve as documentation. SAL allows assertions of the following forms:

1. ASSERTION <number> : <boolean expression>
2. ASSERTION <number> : <boolean expression> FOR EVERY <var>
   FROM <expr> TO <expr>
3. ASSERTION <number> : <boolean expression> FOR SOME <var>
   FROM <expr> TO <expr>

In debugging mode, code is compiled to check the assertions; in optimizing mode no code is compiled. If an assertion is violated in debugging mode, an error message will be printed. Because no code is compiled in optimizing mode, it is not necessary to remove the assertions from the final program. They remain as documentation. Furthermore, if an error occurs later, or if the program is subsequently modified, the assertion checking can be easily enabled again.

The second form of an assertion checks the assertion for all values of the dummy variable. If the assertion fails to hold for any of the specified values, an error message is printed. The third form checks to see that the assertion holds for at least one value in the range. The dummy variable may be the same as a program variable, in which case the value of the program variable will not be affected by the assertion statement.

As an example of the use of assertions, consider a sort procedure that sorts the array A from 1 to 100. After the sort has completed, one
asserts that no value in A is larger than its successor. This can be written:

\[ A[i] \leq A[i+1] \quad \text{for every } i \text{ from 1 to 99} \]

Note that because no code is produced for assertions in optimizing mode, assertions are no substitute for run time checks on the input data. They are intended to detect logical errors in the program. Assertions are always redundant, but redundancy is essential for astronauts and programmers. Thus you never expect an "assertion violated" message to appear; rather, they are a form of self defense against your own errors.

Examples of ASSERTIONS:

\[ \text{ASSERTION 1 : } i < j \text{ AND } k + 6 = m \]
\[ \text{ASSERTION 2 : } f(n) < g(n) \text{ OR } a > \text{OCTAL (111)} \]
\[ \text{ASSERTION 999 : } A[i] = B[i] \text{ for every } i \text{ from } k + 1 \text{ to } n - 6 \]
\[ \text{ASSERTION 1501 : } A[i] = 0 \text{ for some } i \text{ from } 3 \times k + 1 \text{ to } n - 1 \]

11.2 Flow tracing

The entry and exit from procedures, loops and case statements can be dynamically traced from the console switches. When one of the following switches is turned on, messages are printed on the current output device (as specified by OUTDEV) when the corresponding event occurs.

<table>
<thead>
<tr>
<th>switch</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROC entry</td>
</tr>
<tr>
<td>1</td>
<td>RETURN executed</td>
</tr>
<tr>
<td>2</td>
<td>Loop entered or exited</td>
</tr>
<tr>
<td>3</td>
<td>Case statement entered</td>
</tr>
<tr>
<td>4</td>
<td>Variable tracing</td>
</tr>
<tr>
<td>5</td>
<td>Single stepping</td>
</tr>
</tbody>
</table>

These switches may be turned on and off during program execution.
For PROC entries, the actual parameters are printed along with 
the name. For arrays, only the first word is printed. This printing 
is in decimal.

For RETURN statements, the value returned, if any, is printed.

11.3 Variable tracing

The TRACE statement allows the programmer to specify that a line 
is to be printed whenever certain variables are assigned to. Both the 
variable name and its new value are printed. A TRACE statement only causes 
tracing of assignments made within the procedure of the TRACE statement. 
Thus if a variable is passed as a parameter and changed in the called 
procedure, no printout will occur. Local, parameter and external scalars 
may all be traced. Arrays may not be traced. Console switch 4 down disables 
variable tracing.

The UNTRACE statement causes the compiler to stop generating tracing 
code for the variables listed.

Examples:

TRACE A,B,C, HORSE, DOG, PIG, SHEEP

UNTRACE HORSE, PIG, SHEEP

11.4 Single Stepping

This feature provides a method for executing a program one statement 
at a time with a printout of the source statement before each statement is 
executed. This is the high level equivalent of "single cycling" using the 
console switches and lights.

To enable this feature, the statement

SINGLE STEPPING ON

should appear before any executable statements. In this mode, the compiler 
generates a call to the procedure STEP just before each statement. STEP is
provided with one parameter, a character string (byte array) using the
standard SAL calling sequence This character string contains the original
source statement (sometimes truncated).

The user can provide his own version of STEP to print all or some of
the strings. Each string begins with one of the following:

LET, IF, WHILE, DO FOREVER, REPEAT, FOR, CASE, RETURN, PRINT, CALL or EXITLOOP

The standard library procedure STEP, which will be used unless the
user provides his own, works as follows. Before each source statement is
executed, it is displayed on the current output device. Then the PDP-11/45
waits for input from the keyboard. A carriage return will cause it to continue,
execute the statement, display the next statement, and then wait for more
input. The letter "S" followed by a return will call SPIDER (see section
11.5). Console switch 5 DOWN disables single stepping. Typing a line
containing a G will switch input and output to the GT-40.

By providing his own version of STEP, a user can also count the
execution frequency of various statements or statement types, and many
other parameters.

11.5 SPIDER

SPIDER is a general purpose interactive debugging program. It can
be called explicitly by the program, providing a breakpoint facility; it
can be called from the console while single stepping, and it is called
automatically when a run time error is detected. As a breakpoint routine, it
expects a parameter, an integer expression \( \geq 1 \). SPIDER is a symbolic
debugger. It uses the runtime symbol table generated by the compiler in
debugging mode. Using SPIDER, a user can specify the symbolic name of a
variable and have its value typed out in octal or decimal. He can also ask
for a symbolic dump of all variables on the printer, or modify memory or
search memory for various words or combinations of bits, etc. It is also
possible to continue execution from the point where SPIDER was called, or from another place. A complete description of the facilities provided by SPIDER can be found in the SPIDER manual.

11.6 DEBUG statement

While debugging a program, it is common to include statements to print out intermediate results, with the intention of removing them when the program is working. SAL provides a method for including arbitrary statements in a program that will be compiled only in debugging mode, and not in optimizing mode. This eliminates the need for physically removing them from the finished program. It also makes it easy to "reinsert" them if the program ceases to work later.

To use this feature, write

```
DEBUG <group>
GUBED
```

where any number of statements of any type may appear between DEBUG and GUBED. (Nesting DEBUG statements does not make sense).

11.7 Subscript checking

Optional subscript checking is available. If a subscript range error is detected, a message is printed. In debugging mode, the default is subscript checking. In optimizing mode, the default is no subscript checking.

11.8 Procedure call monitoring

In debugging mode, every procedure call is recorded, and a table of all procedures called and how often each one has been called can be obtained by a call to the procedure MONIT. The standard termination procedure,
EXIT, also calls MONIT. The maximum depth of procedure calls and stack size required are all printed. In optimizing mode procedure calls are not recorded.

11.9 Program statistics

In debugging mode, a table of statement types and how often each one occurred in the source program will be printed after translation, if the statement STATISTICS ON appears in the program.

11.10 Enabling and disabling individual debugging features

The various debugging facilities described above are normally included in all programs translated in debugging mode, except STATISTICS and STEPPING, which must be specifically enabled. In optimizing mode, all are suppressed. However, it is possible to override the basic mode (debugging or optimizing) and enable or disable specific kinds of checking. This is done by the following statements, each of which is followed by the word "ON" or "OFF", but not both.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERTION CHECKING</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>FLOW TRACING</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>SINGLE STEPPING</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>SYMBOL TABLE</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>DEBUGGING STATEMENTS</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>SUBSCRIPT CHECKING</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>MONITORING</td>
<td>ON/OFF</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>ON/OFF</td>
</tr>
</tbody>
</table>

The numbers in parentheses are the section numbers in which the feature is described. These names may not be used as variable names if the corresponding statement is used. Nor may they be EQUATED to anything.

Examples:
OPTIMIZE
MONITORING ON
SUBSCRIPT CHECKING ON
PROC P
// OPTIMIZING MODE, EXCEPT THAT SUBSCRIPT CHECKING AND PROC
MONITORING WILL BE PERFORMED;
:
:
END

ASSERTION CHECKING OFF
SYMBOL TABLE OFF
STATISTICS OFF
PROC P1
// DEBUGGING MODE, EXCEPT ASSERTIONS WILL BE IGNORED, NO RUN TIME
SYMBOL TABLE WILL BE GENERATED FOR SPIDER, AND THE STATEMENT
STATISTICS WILL NOT BE PRINTED;
:
:
END
12. Assembly code

Although it is not especially encouraged, MACRO-11 statements may be included anywhere in the source program between \<<\ and \>>. Such statements must not use variables, and should not use registers R3, R4, and R5, nor should SP be changed.

Examples:

\<< SPL 5 >>
\<< RESET >>

IF \textbf{X} > 0

THEN \<< ADD # 1, RO >>
\<< ADC R1 >>

FI

FOR \textbf{I} FROM 1 TO \textbf{N}

DO \<< MFPD (RO) + >>
\<< MOV (SP) +, (R1) + >>

OD
13. **Built-in functions**

The standard SAL library (LIB.OBJ[1,1]) contains the following procedures (except **ADDRESS**, which is compiled in-line)

**ADDRESS**(N)  - Returns the address of the variable N. N must be a scalar variable or array name, not an array element or expression.

**CONT**(N)  - The inverse of **ADDRESS**. N is assumed to contain a pointer (memory address). **CONT** returns the contents of the word pointed to.

**EXIT**  - Print procedure call distribution and return to the operating system.

**INSTR**(LINE)  - A line is read from the current input device, as selected by the EXTERNAL variable INDEV (0=KB:, 1=GT40, 2=PR:) A line is terminated by a car. return. A zero byte is appended. No checking for line overflow is performed. The characters are echoed as typed.

**INSYM**(N)  - One character is read into N from the current input device, as selected by INDEV.

**MONIT**  - A list of all procedures called and their call frequencies is printed.

**NLCR**  - A carriage return and line feed are printed on the current output device.
NULL(N) - N zero bytes are printed on the current output device.

OUTSTR(LINE) - Characters are printed on the current output device until a zero byte is encountered in the text.

OUTSYM(N) - The low order 8 bits of N are printed on the current output device. OUTSYM may be called with an element of a byte array as parameter, i.e. odd addresses are o.k.

PRAD51(N) - N is printed on the current output device as 3 RAD50 characters.


PRNUM(N) - The number in N is printed as a decimal integer in a field 8 characters wide.

PRNUM8(N) - The number in N is printed as an octal integer in a field 8 characters wide.

REM(N,DIV) - The remainder of N/DIV is returned

SKIP(N) - One carriage return and N line feeds are printed on the current output device.
14. Idiosyncrasies

SAL is translated by the ML/I macro processor. This gives rise to various peculiarities listed below. Furthermore, SAL interfaces with DOS, giving rise to still more strange features, to wit:

1. The characters // are used to denote comments. They must not appear in strings.
2. Each expression must be on a single line.
3. Only 1 dimensional arrays are allowed.
4. Except for IF, WHILE, REPEAT, CASE, FOR and DO FOREVER, all statements and declarations must occupy exactly 1 line.
5. The reserved words listed in section 2 must not be used as variable names. (See also section 11.10)
6. All PROC names and EXTERNAL variables are limited to 6 characters, and no 2 PROC's may have the same first 4 letters, nor may any PROC name of 4 or fewer characters be the same as another PROC name.
7. The word FOR and the following expressions must be on the same line. The following DO may or may not be.
8. In a CASE statement, CASE and IN, must be on the same line; the comma following a) must be on the same line as the), and ESAC must be alone on a line. The clause number must be on the same line as its clause.
9. Procedures may not be declared within procedures.
10. Arithmetic expressions are evaluated strictly left to right, with no priority rules, and no parentheses allowed.
11. A line, in the above sense, may have 1 or more continuation lines.
15. Operating instructions.

The modules of a SAL program can and must be individually translated. The translator accepts one SAL procedure module, or one SAL data module, and produces a program for MACRO-11, which is then assembled by MACRO-11 to a relocatable binary module. A collection of relocatable binary modules can be linked and loaded by the DOS loader LINK-11.

Independent translation of modules is extremely important. It means that if one module must be changed, it is not necessary to retranslate all modules. For a program with hundreds or thousands of procedures, this can make the difference between success and failure of the entire project. Note that the ALGOL static scope rules (block structure) make independent compilation of procedures difficult if not impossible. It is not possible to compile the ALGOL procedure

```
procedure sum (a,b) ; a := b+c;
```

separately from the rest of the program because

1. the compiler does not know if a fixed or floating point add is needed.
2. the compiler does not know how to access the global variable c. The access method depends upon the block c was declared in, and when compiling the above procedure, the compiler does not know where c was declared.

One consequence of this is that ALGOL is never used for "serious" programs i.e. large programs (100 pages and up) that will actually be run, (as opposed to university programs which are written to demonstrate something but are not used to solve some problem that someone wants solved).

To translate a SAL module, the module should be put on a file all by itself, with no other modules. In the example below the name PROG.SAL will be used as the name of the input file. The relocatable binary will be put
PROG is just an example. The extension SAL should be used for all SAL modules.

15.1 Translation SAL procedure modules

1. Type: RUN SAL

2. Wait for the message: SAL V006

3. Type your input file name, e.g. PROG.SAL or DECLS, PROG.SAL

4. Wait for the message: SALMAC V001

   This means translation is finished, and that SALMAC has been loaded.

5. To assemble the generated program, type a command string to SALMAC

   If your source file is PROG.SAL type: PROG<A,B

   If you want an assembly listing: PROG, LP:<A,B

6. When SALMAC finishes, your relocatable binary file is ready.

15.2 Translation of SAL DATA SEGMENTS

   Same as procedure modules, except use SALDS instead of SAL.

15.3 Execution of a collection of modules:

   To link the modules A.OBJ, B.OBJ, C.OBJ and begin execution with A:

1. Type: RUN LINK

2. Type: XYZ<A,B,C,LIB[1,1] /L/ TR:A/G0

   where XYZ is the name of the load file (load module). Note:

   LIB is a library.

   The name after /TR: is the starting address. This proc must have 0

   parameters.

   If any of the PROC’s had its name on the END statement, the /TR switch is not
16. Error messages

SAL detects a variety of errors and complains about them. Each error is identified by a number and a message. An explanation of the errors is given below.

<table>
<thead>
<tr>
<th>error</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Undefined symbol. Typing error or missing declaration.</td>
</tr>
<tr>
<td>2.</td>
<td>The specified name is being used in a syntactic position where a table name is expected. It has not been declared in a TABLE declaration.</td>
</tr>
<tr>
<td>3.</td>
<td>OF followed by a table name is not followed by an open square bracket.</td>
</tr>
<tr>
<td>4.</td>
<td>Something strange follows an array element.</td>
</tr>
<tr>
<td>5.</td>
<td>Right parenthesis missing in a CALL or function.</td>
</tr>
<tr>
<td>6.</td>
<td>Table or field name has occurred in a position where a variable is expected.</td>
</tr>
<tr>
<td>7.</td>
<td>Field name has not been declared (detected after a scalar)</td>
</tr>
<tr>
<td>8.</td>
<td>Array name has been used without a subscript.</td>
</tr>
<tr>
<td>9.</td>
<td>For loop with a negative BY part has no DOWNTO part.</td>
</tr>
<tr>
<td>10.</td>
<td>Assertion number is incorrect</td>
</tr>
</tbody>
</table>
11. Controlled variable in a FOR loop is not a scalar.

12. Field name has not been declared. (detected after an array element).

13. Subscript contains a term other than a register or scalar variable.

14. Subscript contains an operator other than + - *.

15. Closing square bracket directly followed by a term, rather than by an operator.

16. Two consecutive operators detected.

17. Right parenthesis directly followed by a term, rather than by an operator.

18. Missing expression, or leading sign other than + or -.

19. Procedure name has been declared as a variable, field, or table.

20. Parameter in a call or function does not conform to the rules for actual parameters, or is undefined.

21. Operator other than + - or * used in a simple expression passed as an actual parameter.

22. Term other than a register or scalar variable used in a simple expression passed as an actual parameter.
23. Actual parameter not followed by comma or right parenthesis.

24. Illegal operator detected in an expression.

25. Something other than a scalar variable is listed in an TRACE statement.

26. Something other than a scalar variable is listed in an UNTRACE statement.

27. Attempt to assign a value to a constant.

28. Function call used on left hand side of an ASSIGNMENT.

29. Bit number in a FIELD declaration is non numeric.

30. Incorrect bit number in a FIELD declaration. Both bit numbers must be between 0 and 15 inclusive, and the first must be smaller than or equal to the second.

31. A procedure has been declared LOCAL.

32. Syntax error in an EXTERNAL declaration, or attempt to declare as external a name not conforming to the rules for names.

33. Syntax error in a LOCAL declaration, or attempt to declare a local variable whose name is improper.

34. FIELD name incorrect.

35. Attempt to enable or disable a debugging feature does not contain
36. CASE clause not preceded by a number or the word OUT.

37. CASE clause whose number is not consecutive with previous CASE clause detected.

38. Syntax error in a formal parameter list.

39. Syntax error in a TABLE declaration.

40. Non-numeric bound in the declaration of an external array.

41. Non-numeric bound in the declaration of a local array.

42. Non-numeric bound in a formal parameter array.

43. The declaration list of a STATIC declaration requires more space than the array has.
17. Context Free Grammar for SAL

Layout (spaces, new lines, etc.) have been ignored in this grammar. Nevertheless they are significant.

The superscripts are cross references, specifying where the non terminal is defined.

17.1 Program structure

1. \( <\text{program}> ::= <\text{module}>^2 \mid <\text{module}>^2<\text{program}>^1 \)
2. \( <\text{module}> ::= <\text{procedure module}>^3 \mid <\text{data module}>^{80} \)

17.2 Procedure structure

3. \( <\text{procedure module}> ::= <\text{prefix}>^6 <\text{header}>^4<\text{tail}>^{22} \mid <\text{header}>^4<\text{tail}>^{22} \)
4. \( <\text{header}> ::= <\text{proc decl}>^5 \mid <\text{proc decl}>^5 <\text{decl part}>^8 \)
5. \( <\text{proc decl}> ::= \text{PROC} <\text{name}>^{90} \mid \text{PROC} <\text{name}>^{90} (<\text{decl list}>^{13}) \)
6. \( <\text{prefix}> ::= <\text{prefix statement}>^7 \mid <\text{prefix}>^6<\text{prefix statement}>^7 \)
7. \( <\text{prefix statement}> ::= \text{OPTIMIZE} \mid <\text{equate decl}>^{21} \mid <\text{option}>^{104} \)

17.3 Declarations

8. \( <\text{decl part}> ::= <\text{decl}>^9 \mid <\text{decl}>^9<\text{decl part}>^8 \)
9. \( <\text{decl}> ::= <\text{variable decl}>^{10} \mid <\text{other decl}>^{11} \)
10. \( <\text{variable decl}> ::= <\text{local decl}>^{12} \mid <\text{external decl}>^{17} \)
11. \( <\text{other decl}> ::= <\text{field decl}>^{18} \mid <\text{table decl}>^{19} \mid <\text{equate decl}>^{21} \)
12. \( <\text{local decl}> ::= \text{LOCAL} <\text{decl list}>^{13} \)
13. \( <\text{decl list}> ::= <\text{decl item}>^{14} \mid <\text{decl item}>^{14}, <\text{decl list}>^{13} \)
14. \( <\text{decl item}> ::= <\text{name}>^{90} \mid <\text{word array}>^{15} \mid <\text{byte array}>^{16} \)
15. \( <\text{word array}> ::= <\text{name}>^{90} [<\text{integer}> : <\text{integer}>] \)
16. \( <\text{byte array}> ::= <\text{name}>^{90} [<\text{integer}>^{102} : <\text{integer}>^{102}] \text{BYTE} \)
17. \[\text{<external decl> ::= EXTERNAL <decl list>}\]
18. \[\text{<field decl> ::= FIELD <name> = BITS <bitno> TO <bitno>}\]
19. \[\text{<table decl> ::= TABLE <name list>}\]
20. \[\text{<name list> ::= <name> | <name>, <name list>}\]
21. \[\text{<equate decl> ::= EQUATE <name> = <string>}\]

17.4 Procedure bodies

22. \[\text{<tail> ::= <group> | <end statement> | <end statement>}\]
23. \[\text{<group> ::= <statement> | <statement> <group>}\]
24. \[\text{<statement> ::= <control statement> | <one liner> | <debug group>}\]
25. \[\text{<control statement> ::= =<loop> | <chooser>}\]
26. \[\text{<loop> ::= <while> | <repeat> | <for> | <do forever>}\]
27. \[\text{<chooser> ::= <if> | <case>}\]
28. \[\text{<one liner> ::= <basic> | <trans> | <debug>}\]
29. \[\text{<basic> ::= <let> | <print>}\]
30. \[\text{<trans> ::= <call> | <return> | <exitloop>}\]
31. \[\text{<debug> ::= <assertion> | <tracer>}\]
32. \[\text{<end statement> ::= END | END <name>}\]

17.5 Control Statements

33. \[\text{<while> ::= WHILE <expression> DO <group> OD}\]
34. \[\text{<repeat> ::= REPEAT <group> UNTIL <expression> LITNU}\]
35. \[\text{<for> ::= FOR <name> <count part> DO <group> OD}\]
36. \[\text{<count part> ::= FROM <expression> <directionby>}\]
37. \[\text{<directionby> ::= =<to part> | <by part>}\]
38. \[\text{<to part> ::= TO <expression> | DOWNTO <expression>}\]
39. \[\text{<by part> ::= BY <expression>}\]
40. \[\text{<do forever> ::= DO FOREVER <group> OD}\]
17.6 One liners

48. <let> ::= LET <destination>66 = <expression>61
49. <print> ::= PRINT <print list>50
50. <print list> ::= <print item>51 | <print item>51, <print list>50
51. <print item> ::= <expression>61 | <string>97
52. <call> ::= CALL <name>90 | CALL <name>90 <parameter part>53
53. <parameter part> ::= () | (<parameter list>54)
54. <parameter list> ::= <parameter>55 | <parameter>55, <parameter list>54
55. <parameter> ::= <simple expr>73 | <string>97 | <array element>88
56. <return> ::= RETURN | RETURN <expression>61 | RETURN FROM INTERRUPT
57. <exitloop> ::= EXITLOOP
58. <assertion> ::= ASSERTION <unsigned integer>101 : <asstail>59
59. <asstail> ::= <expression> | <expression> FOR EVERY <name> FROM <expression> TO <expression> | <expression> FOR SOME <name> FROM <expression> TO <expression>
60. <tracer> ::= TRACE <namelist>20 | UNTRACE <namelist>20

17.7 Expressions

61. <expression> ::= <relation>62 | <relation>62 <bool op>78 <expression>61
62. <relation> ::= <arith expr>63 | <arith expr>63 <rel op>11 <arith expr>63
63. <arith expr> ::= <plain ae>64 | + <plain ae>64 | - <plain ae>64
64. \( \langle \text{plain ae} \rangle ::= \langle \text{term} \rangle^6 \langle \text{term} \rangle^5 \langle \text{op} \rangle^7 \langle \text{plain ae} \rangle^6 \)
65. \( \langle \text{term} \rangle ::= \langle \text{destination} \rangle^6 \langle \text{unsigned integer} \rangle^10 | \langle \text{function call} \rangle^79 \)
66. \( \langle \text{destination} \rangle ::= \langle \text{name} \rangle^9 | \langle \text{piece} \rangle^6 \)
67. \( \langle \text{piece} \rangle ::= \langle \text{array element} \rangle^6 \langle \text{selection} \rangle^5 \langle \text{bit field} \rangle^72 \)
68. \( \langle \text{array element} \rangle ::= \langle \text{name} \rangle^9 [\langle \text{simple expr} \rangle^72] \)
69. \( \langle \text{selection} \rangle ::= \langle \text{primary} \rangle^70 \langle \text{secondary} \rangle^71 \)
70. \( \langle \text{primary} \rangle ::= \langle \text{name} \rangle^9 \text{OF} | \langle \text{name} \rangle^9 [\langle \text{simple expr} \rangle^73] \text{OF} \)
71. \( \langle \text{secondary} \rangle ::= \langle \text{name} \rangle^9 [\langle \text{simple expr} \rangle^73] \)
72. \( \langle \text{bit field} \rangle ::= \langle \text{name} \rangle^9 . \langle \text{name} \rangle^9 | \langle \text{array element} \rangle^5 . \langle \text{name} \rangle^9 \)
73. \( \langle \text{simple expr} \rangle ::= \langle \text{term} \rangle^74 | \langle \text{term} \rangle^74 \langle \text{simple op} \rangle^75 \langle \text{simple expr} \rangle^73 \)
74. \( \langle \text{term} \rangle ::= \langle \text{name} \rangle^9 | \langle \text{unsigned integer} \rangle^101 \)
75. \( \langle \text{simple op} \rangle ::= \text{-} | \text{x} | \text{y} \)
76. \( \langle \text{op} \rangle ::= \text{+} | \text{-} | \text{*} | \text{/} | \text{&} | \text{|} | \text{#} \)
77. \( \langle \text{rel op} \rangle ::= \text{<} | \text{=} | \text{!=} | \text{>} | \text{>=} \)
78. \( \langle \text{bool op} \rangle ::= \text{AND} | \text{OR} \)
79. \( \langle \text{function call} \rangle ::= \langle \text{name} \rangle^9 \langle \text{parameter part} \rangle^53 \)

17.8 Data segments

80. \( \langle \text{data module} \rangle ::= \langle \text{data header} \rangle^81 \langle \text{data decl part} \rangle^82 \text{END} \)
81. \( \langle \text{data header} \rangle ::= \text{DATA SEGMENT} \langle \text{octit} \rangle^95 \)
82. \( \langle \text{data decl part} \rangle ::= \langle \text{data decl} \rangle^83 | \langle \text{data decl} \rangle^83 \langle \text{data decl part} \rangle^82 \)
83. \( \langle \text{data decl} \rangle ::= \langle \text{external decl} \rangle^17 | \langle \text{static decl} \rangle^84 | \langle \text{equate decl} \rangle^21 \)
84. \( \langle \text{static decl} \rangle ::= \text{STATIC} \langle \text{static list} \rangle^85 \)
85. \( \langle \text{static list} \rangle ::= \langle \text{static item} \rangle^86 | \langle \text{static item} \rangle^86 \langle \text{static list} \rangle^85 \)
86. \( \langle \text{static item} \rangle ::= \langle \text{decl item} \rangle^14 | \langle \text{initialized} \rangle^87 \)
87. \( \langle \text{initialized} \rangle ::= \langle \text{name} \rangle^90 \langle \text{initialized} \rangle^88 | \langle \text{word array} \rangle^15 \)
88. \( \langle \text{initialized} \rangle^89 \langle \text{initial list} \rangle^89 \)
89. \( \langle \text{initial list} \rangle ::= \langle \text{initial} \rangle^88 | \langle \text{initial} \rangle^88 \langle \text{initial list} \rangle^89 \)
17.9 Miscellaneous

90.  \(<\text{name}> ::= \langle\text{letter}\rangle^93 \mid \langle\text{letter}\rangle^93 \langle\text{name tail}\rangle^91\rangle

91.  \langle\text{name tail}\rangle ::= \langle\text{name char}\rangle^92 \mid \langle\text{name char}\rangle^92 \langle\text{name tail}\rangle^91

92.  \langle\text{name char}\rangle ::= \langle\text{letter}\rangle^93 \mid \langle\text{digit}\rangle^94 \mid \langle\text{underline}\rangle^96

93.  \langle\text{letter}\rangle ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

94.  \langle\text{digit}\rangle ::= 0|1|2|3|4|5|6|7|8|9

95.  \langle\text{octit}\rangle ::= 0|1|2|3|4|5|6|7

96.  \langle\text{underline}\rangle ::= _

97.  \langle\text{string}\rangle ::= "<\text{no quote}\rangle^98"

98.  \langle\text{no quote}\rangle ::= <\text{char}\rangle^99 \mid \langle\text{char}\rangle^99 \langle\text{no quote}\rangle^98

99.  \langle\text{char}\rangle ::= <\text{name char}\rangle^92 \mid \langle\text{special}\rangle^100

100. \langle\text{special}\rangle ::= \langle\text{op}\rangle^76 | ([|(|]|.|<|>|=|=|/|:|;|?|~|%|\',|\"

101. \langle\text{unsigned integer}\rangle ::= \langle\text{digit}\rangle^94 \mid \langle\text{digit}\rangle^94 \langle\text{unsigned integer}\rangle^101

102. \langle\text{integer}\rangle ::= \langle\text{unsigned integer}\rangle^101 \mid + \langle\text{unsigned integer}\rangle^101

- \langle\text{unsigned integer}\rangle^101

103. \langle\text{bit no}\rangle ::= \langle\text{digit}\rangle^94|10|11|12|13|14|15

17.10 Options

104. \langle\text{option}\rangle ::= \langle\text{option type}\rangle^105 \langle\text{option type}\rangle ^ OFF

105. \langle\text{option type}\rangle ::= \langle\text{assertion checking}\rangle \mid \langle\text{flow tracing}\rangle

SINGLE STEPPING \mid \langle\text{symbol table}\rangle

DEBUGGING STATEMENTS \mid \langle\text{SUBSCRIPT CHECKING}\rangle

MONITORING \mid \langle\text{STATISTICS}\rangle

Note: vertical bar is also a special
PROC TOWERS(K, POLE1, POLE2, POLE3)

// THIS PROCEDURE SOLVES THE TOWERS OF HANOI;
LOCAL M
IF K=1
THEN PRINT "MOVE DISK FROM 'POLE1, "TO", POLE2"
ELSE LET M=K-1
    CALL TOWERS(M, POLE1, POLE3, POLE2)
    CALL TOWERS(1, POLE1, POLE2, POLE3)
    CALL TOWERS(M, POLE3, POLE2, POLE1)
FI
END

PROC SORT (R[1:10])

// THIS PROCEDURE PERFORMS A SORT USING THE INTERCHANGE SORT
ALGORITHM IT ALSO PRINTS THE ARRAY BEFORE AND AFTER SORTING;
EQUIATE I= R0
EQUIATE J = R1

//FIRST PRINT THE ARRAY BEFORE SORTING IT;
FOR I FROM 1 TO 10 BY 1 DO PRINT I, R[I] OD
// INTERCHANGE SORT;

FOR I FROM 1 TO 9 BY 1
    DO FOR J FROM I+1 TO 10 BY 1
        DO IF R[I] > R[J]
            THEN LET R2 = A[I]
        OD
    OD

END
LET A[J] = R2
FI
OD
OD

//PRINT RESULTS;

FOR I FROM 1 TO 10 BY 1 DO PRINT I, A[I] OD
END

PROC SEARCH (A[1:100], N1, N2, KEY)

//BINARY SEARCH OF A FROM N1 TO N2 LOOKING FOR KEY, KNOWN TO BE IN
THE TABLE;

LOCAL N

EQUATE MIDDLE = RO

LET MIDDLE = N1 + N2/2//NOTE: THIS MEANS (N1+N2)/2;

IF KEY = A[MIDDLE] THEN RETURN MIDDLE FI

IF KEY < A[MIDDLE]
    THEN LET N = MIDDLE -1
        RETURN SEARCH(A, N1, N, KEY)
    ELSE LET N = MIDDLE + 1
        RETURN SEARCH (A, N, N2, KEY)
FI
END
INDEX

ALGOL 3, 8f, 11, 19, 34f, 37f, 53
APL 19
ARRAY 4, 7, 17
   byte 6f, 24
ASC 17f
ASSEMBLY CODE 49
ASSERTION 42f

BLISS 1
   boolean, see expression
   branch 41
   breakpoint 45

Calling sequence 33f
comments 40
continuation 3

data 6
   data segment 37
debugging 42ff, 47f
declaration 7
   EQUATE 15f
   EXTERNAL 8ff
FIELD 14
LOCAL 7f
STATIC 38
TABLE 11ff

device register 9f
DOS 4, 8, 54

field
   bit 14, 17, 24
function call 17f, 50f
GRAMMAR 59ff

identifier 7
idiosyncracy 52
integer
   decimal 17
   octal 17
jump 41

line 3
Macro 12ff, 15
MACRO-11 49
module 3

operand 17
operating instructions 53ff
operator 18
OPTIMIZATION 41
overflow 19

parameter
   actual 18, 22, 32f
   formal 4, 33
parentheses 18, 20
PL/360 1
Priority
   operator 19
procedure 3f, 37

register 17f
reserved word 3, 9f, 47

FALSE 22