**SUMMARY**  A definition of the Intermediate Code generated by the first pass of the 4-50 ALGOL Compiler
Introduction

The purpose of this document is to define the form of the Intermediate Code generated by the first pass of the System 4-50 ALGOL Compiler. Since the design of the Intermediate Code is based on that of the Object Program generated by the Whetstone KDF9 ALGOL Translator, use is made of the fact that the Whetstone Object Code is fully defined in the book 'ALGOL 60 Implementation', by B. Randell and L.J. Russell, Academic Press, 1964. The differences between the Intermediate Code and the Whetstone Object Code arise from (i) the three-pass structure of the System 4-50 ALGOL Compiler, which is designed to generate machine code as the end product, and (ii) the altered method of handling the run-time stack, now based on the ideas of the ALCOR-ILLINOIS 7090 ALGOL Compiler.

This document has a scope restricted to a definition of those parts of the System 4-50 ALGOL Intermediate Code which differ from the corresponding parts of the Whetstone KDF9 ALGOL Object Code. A working knowledge of the Whetstone Object Code is therefore assumed, although for those without access to the reference quoted above, a brief introductory section to the System 4 form is included. No attempt will be made in this document to describe the generation of Intermediate Code from the source Algol or the translation of the Intermediate Code into machine code, nor will the detailed modus operandi of the generated machine code programs be mentioned. A brief description of the overall structure of the compiler is included.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTRODUCTION</strong></td>
<td></td>
</tr>
<tr>
<td>A. GENERAL DESCRIPTION OF THE COMPILER.</td>
<td>4</td>
</tr>
<tr>
<td>A(i) Note on the running of the object code.</td>
<td>6</td>
</tr>
<tr>
<td>1. INTRODUCTION TO THE INTERMEDIATE CODE: NOTATION</td>
<td></td>
</tr>
<tr>
<td>1A. Labels</td>
<td>9</td>
</tr>
<tr>
<td>1B. Support Tables</td>
<td>10</td>
</tr>
<tr>
<td>2. BRIEF ILLUSTRATION OF THE INTERMEDIATE CODE IN USE.</td>
<td>13</td>
</tr>
<tr>
<td>2A. Note on the Translation of Designational Expressions.</td>
<td>15</td>
</tr>
<tr>
<td>3. BLOCKS</td>
<td>16</td>
</tr>
<tr>
<td>4. PROCEDURES</td>
<td></td>
</tr>
<tr>
<td>4A. Procedure Structure in the Intermediate Code.</td>
<td>17</td>
</tr>
<tr>
<td>4B. Parameter List Operations.</td>
<td>17</td>
</tr>
<tr>
<td>4C. Calls of Function Designators and Procedure Statements.</td>
<td>18</td>
</tr>
<tr>
<td>4D. Actual Operations.</td>
<td>19</td>
</tr>
<tr>
<td>4E. Example of a Procedure Call.</td>
<td>20</td>
</tr>
<tr>
<td>4F. Use of Formal Parameters within a Procedure Body.</td>
<td>22</td>
</tr>
<tr>
<td>5. SWITCHES</td>
<td></td>
</tr>
<tr>
<td>5A. Switch Structure in the Intermediate Code.</td>
<td>24</td>
</tr>
<tr>
<td>5B. Switch Designators and Array Elements as Actual Parameters.</td>
<td>26</td>
</tr>
<tr>
<td>6. CONDITIONAL STATEMENTS AND EXPRESSIONS</td>
<td>27</td>
</tr>
<tr>
<td>7. ARRAYS</td>
<td></td>
</tr>
<tr>
<td>8. FOR STATEMENTS</td>
<td></td>
</tr>
<tr>
<td>8A. General</td>
<td>30</td>
</tr>
<tr>
<td>8B. For Statement Structure in the Intermediate Code.</td>
<td>30</td>
</tr>
<tr>
<td>8C. Arithmetic Element</td>
<td>32</td>
</tr>
<tr>
<td>8D. While Element</td>
<td>33</td>
</tr>
<tr>
<td>8E. Step Until Element</td>
<td>34</td>
</tr>
<tr>
<td>8F. Example of a For Statement.</td>
<td>35</td>
</tr>
<tr>
<td>9. OWN VARIABLES</td>
<td></td>
</tr>
<tr>
<td>10. NOTE ON INDEPENDENT COMPILATION OF ALGOL PROCEDURES.</td>
<td>38</td>
</tr>
<tr>
<td>11. LIBRARY PROCEDURES</td>
<td></td>
</tr>
<tr>
<td>12. THE INTERMEDIATE CODE OPERATION TRACE</td>
<td>41</td>
</tr>
<tr>
<td>13. THE INTERMEDIATE CODE OPERATIONS FAIL AND LPAIL.</td>
<td>43</td>
</tr>
</tbody>
</table>
14. DICTIONARY OF INTERMEDIATE CODE OPERATIONS. 44
14A. Parameter Notation. 48
14B. Hexadecimal Codes. 50
15. 4-50 ALGOL HARDWARE REPRESENTATION 51
A. GENERAL DESCRIPTION OF THE COMPILER

The System 4-50 ALGOL compiler is a three pass compiler, which translates source ALGOL into machine code modules. The three passes of the compiler are known as Phase 10, Phase 20, Phase 30. The System 4-50 ALGOL language and the compiler's relationship with the 5J Program Trials System are fully described in the System 4-50 ALGOL Reference Manual, Parts I and II.

The compiler translates a module at a time. A source ALGOL program, or a procedure for independent compilation, is presented to Phase 10. The action of Phase 10 is to

10.a) convert the source ALGOL into an internal code called Intermediate Code, generating as it does so

10.b) a set of two support tables. One is the Position Identifier Table, and is a reference table of labels and procedures, the other is the Level Parameters (b), containing information about the types and characteristics of procedures and their parameters. The Intermediate Code and the Position Identifier Table may be listed on request, as may

10.c) the Storage Map, a listing of the stack locations of the ALGOL variables. This item is not preserved after listing. A further function of Phase 10 is

10.d) to perform rigorous checks on the syntactic legality of the source ALGOL. The failure mechanism employed is, briefly, to detect and notify as many syntactic errors as possible without aborting either the translation or the generation of Intermediate Code and support tables. Upon detecting a syntactic error, the error is notified and a FAIL operation is planted in the generated Intermediate Code; then error recovery procedures are used to find the earliest point in the succeeding text at which translation can be resumed. This generally implies the loss of a statement or of a cluster of declarations. Naturally, this mechanism is not applied to catastrophic failures. The result is a sprinkling of FAIL operations in the generated Intermediate Code for the various syntactic errors detected. At object execute time each of these is an entry into the run-time failure routine, so that the first one to be dynamically encountered will terminate the run.

Phase 10 writes the Intermediate Code (10.a) and the support tables (10.b) to a work file, and finally initiates the call of Phase 20 as an overlay to itself. The action of Phase 20 is to

20.a) process the Intermediate Code and the support tables produced by Phase 10, to generate machine code with forward references missing.

20.b) The information required to supply these missing references is built up into a table which is an updated machine address form of the Position Identifier Table.

Phase 20 writes the machine code in this form (20.a) and the reference table (20.b) to a work file, and finally initiates the call of Phase 30 as an overlay to itself.
The action of Phase 30 is to

30.a) update the machine code using the information left in the work file by Phase 20. Phase 30 also

30.b) ensures that the resulting machine code module is in a form recognisable to the operating regime in which it will eventually run. This involves the generation of ESD, RLD, and END information. Then

30.c) the generated machine code module may optionally be listed in pseudo-usercode form.

The organisation of the transfer of information between one pass and the next via the work file (on the 5J Trials Library disc) is performed by a set of three special purpose disc handling routines (D.H.R.s) one for each pass. These D.H.R.s are written so as to use the disc as efficiently as possible.

The overlay structure of the compiler is controlled by a small root segment which consists of a single control module. The overlays brought down successively by the root segment are:

(1) Options processor
(2) Phase 10
(3) Phase 20
(4) Phase 30

Each overlay returns control to ROOT which then brings down the next overlay or returns control to the system (Compilation Control).
A(i). Note on the Running of the Object Code.

A relocatable binary module generated by the compiler will contain external references to other modules, namely:

a) independently compiled modules, written originally in ALGOL or in usercode (later extensions may be made to include other languages), written by the user and called by him in the module concerned. If written in usercode, the rules listed in the Part II Reference Manual must be observed.

b) a standard module without which no object program can be run, containing a set of support routines for processes initiated in the object module or bound object modules being run. These routines are the Slave Routines. Examples are Float, Fix, Call by name, Procedure entry, Make array storage function, etc. The full list appears in the Part II Reference Manual. This module is written initially in usercode, and is a standard package, supplied with the compiler.

c) a set of input-output modules, in sufficient number to satisfy the peripheral requirements of the program being run. The input-output procedures required to use the full range of System 4-50 standard peripheral devices are arranged in self-contained groups. The arrangement into modules is fully described in the Part II Manual. The input-output modules are written initially in usercode, and are supplied with the compiler as standard packages.

d) if required, any of the standard function modules required to perform the functions SIN, COS, etc. For a full list see the Reference Manual Part II. These modules are written initially in usercode, and are supplied with the compiler as standard packages.

e) a dummy module destined to hold the run-time stack. This must always appear in store after all the other modules, that is its start address must be higher than the final address of any other module, to allow for its expansion when the object program is running. This module must always be present.

Having bound the generated object module with such object modules of types (a), (b), (c), (d) and (e) as are necessary, the resulting program may be run in the normal way. Note that modules of types (b), (c) and (d) will be bound automatically into the root segment unless otherwise specified. It is hoped to arrange for the stack (e) to be automatically included in the correct position for programs which do not use overlays.
1. INTRODUCTION TO THE INTERMEDIATE CODE:

NOTATION

The Intermediate Code generated by Phase 10 is a string of variable length pseudo-instructions, each pseudo-instruction comprising an operation field of 1 byte, and an operand field of length zero bytes or greater.

The operation field for convenience is assigned a mnemonic name for use in discussion. Such mnemonics are CF ("Call Function"), TFI ("Take Formal Integer"), PST ("Parameter String"). For the full list, see the Dictionary of Intermediate Code Operations, section 14. For the hexadecimal internal representations of these operations, see section 14B.

The operand field may consist of counts, labels, pointers, stack addresses, constants, or may be absent. The symbols used in this Report to denote quantities in the operand field are:

- **k** The hierarchy number. This is the nested procedure depth of the current procedure, starting at zero for the outermost program level. The maximum permitted value is \( k = 14 \).

- **s** The stack address, in words, within the hierarchy, starting at \( s = 0 \).
  
  Both \( k \) and \( s \) are needed to specify a stack address completely. Stack addresses are denoted by \((k, s)\), indicating that \( k \) and \( s \) are packed into two bytes, with 4 bits for \( k \) and 12 bits for \( s \).

- **n** The block level within the current (or destination) hierarchy, starting at \( n = 1 \). \( n \) requires 1 byte.

- **\( \pi \)** The \( \pi \)th entry in a support table called Level Parameters (b) (see below section 1B). Each entry in this table is 1 byte in length. \( \pi \) requires two bytes.

- **m** Number of words of stack space in the current hierarchy required to hold the link data and parameters for this hierarchy. \( m \) requires 1 byte.

- **N** The maximum block nesting depth within the current hierarchy, that is the greatest value of \( n \). \( N \) requires 1 byte.

- **a** Number of arrays in an array segment. \( a \) requires 1 byte.

- **b** Number of dimensions of an array. \( b \) requires 1 byte.

- **r** The \( r \)'th failure in this module. \( r \) requires 1 byte.

- **t** The type of the operation which eventually gave rise to a syntactic failure. \( t \) requires 1 byte.

- **S4** Special form of the parameter \( s \); thus \((k, S4)\) gives the stack address of the first of the locations in the stack assigned to a for statement.
identifier An identifier of up to eight EBCDIC characters, left hand justified and padded with spaces (X'40') at the right-hand end if necessary. Requires 8 bytes.

constant The value of an explicit constant. Requires 4 bytes for integer constants, 8 bytes for real constants.

string A string of basic symbols representing one-for-one the characters in the string as written in the ALGOL program, including all string quotes. One byte for each basic symbol.

L Number of words of stack space in the current hierarchy required to hold the first order working storage, given by 
\[ L = m + \text{number of words of stack space required to hold all variables declared within the hierarchy, including 4 words for every array declaration.} \]

L appears in combined with k, as (k, L), so that 12 bits are required for L.

- A one byte space.

The remaining parameters refer to labels, described in the next section (1A).
1A. **Labels.**

Three kinds of labels are used in the Intermediate Code, as follows:

(i) **L labels.**

Labels written explicitly in the source ALGOL program are referred to internally as L labels. Thus L1 is the (1+1)th user's label to be declared. The number 1 is zero for the first label. As a parameter, l always appears in combination with k in the form (k,l), so that l requires 12 bits. Note that switches are assigned L labels, the name used in the Position Identifier Table (see below Section 1B) being the name of the switch.

(ii) **P labels.**

At a procedure declaration, the procedure is assigned a P label of the form Pp, denoting the (p+1)th procedure to be declared. The number p is zero for the first procedure declared. As a parameter, p always appears in combination with k in the form (k,p), so that 12 bits are required for p. The name used for each P label in the Position Identifier Table (see below) is the name of the procedure.

(iii) **G labels.**

For transfers of control in the Intermediate Code which are neither explicit (that is using a 'GOTO' statement, which employs L labels) nor procedure calls (which use P labels), G labels are generated of the form Gg, where the number g starts from zero for the first. As a parameter, g always appears on its own, and requires two bytes. For a G label, the name part in the Position Identifier Table (see below) is blank.

Throughout this document these labels, in both their left-hand and right-hand incarnations, have been written explicitly into the Intermediate Code for clarity of exposition. In fact, left-hand labels have no existence in the Intermediate Code; they exist only as entries in the Position Identifier Table (see below, Section 1B). Right-hand labels in the Intermediate Code are in fact the quantities l, p, and q defined above. These act as pointers to the appropriate entries in the Position Identifier Table, wherein are found the Intermediate Code addresses to which the labels refer.
1B. Support Tables.

The support tables required with the Intermediate Code are the Position Identifier Table and Level Parameters (b). There is a table called Level Parameters (a), but this is purely internal to Phase 10 and is not relevant to this discussion of the Intermediate Code.

Position Identifier Table:

<table>
<thead>
<tr>
<th>EXTERNAL NAME</th>
<th>INTERNAL NAME</th>
<th>I/C BYTE NO.</th>
<th>CARD NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

External Name: A string of eight 8-bit basic symbols packed into eight 6-bit characters. All external (source) names are held in this form in Phase 10 for economy of space. This item exists only within Phase 10 for use in failure messages and in diagnostic listings.

Internal Name: The first byte contains the 8-bit basic symbol for G, L, or P. The last two bytes contain the numeric value (as a binary integer) of g, l, or p, starting from 0.

I/C Byte No.: The byte number of the labelled intermediate code operation, starting from 1.

Card No.: The current card number as read from the card, held as eight packed decimal digits held in 5 bytes. This item exists only within Phase 10 for use in failure messages and in diagnostic listings.
Level Parameters (b):

\[ P = \text{PROCEDURE} \]
\[ \Pi = \text{PROCEDURE} \]
\[ \Pi = \text{SPECIFICATION END MARKER} \]

Each entry in list (b) is one byte long. The parameter \( \Pi \) (see (1)) is a pointer to a procedure entry in list (b), as indicated in the diagram above. The list of values of \( \Pi \) is internal to Phase 10, and is indexed by the value of \( p \) assigned to each new procedure, including formal procedures. This list is called Level Parameters (a).

The entries in list (b) are, for the various situations:

1) Procedure entry:

\[
\begin{array}{l}
N \neq 00000 \\
a) \ N \text{ is } 1 \text{ if the procedure has parameters, } 0 \text{ otherwise} \\
b) \ The \ next \ 3 \ bits \ are \ always \ zero. \\
c) \ tttt \ gives \ the \ type \ of \ the \ procedure \ according \ to \ the \ following \ scheme: \\
0001 \ procedure \\
1101 \ integer \ procedure \\
1101 \ real \ procedure \\
1011 \ boolean \ procedure.
\end{array}
\]

2) Formal parameter:

\[ Ntttttt \]
a) \( N \) is set to 1 for parameters called by value. Thus \( N \) may be set to 1 for real, integer, boolean, real array, integer array, boolean array, or label parameters.

b) The next seven bits \((t)\) give the type of the parameter according to the following scheme:

\[
\begin{align*}
\emptyset\emptyset11101 & \quad \text{real} & 0111101 & \quad \text{real procedure}\ast \\
0011110 & \quad \text{integer} & 0111110 & \quad \text{integer procedure}\ast \\
0011011 & \quad \text{boolean} & 0111011 & \quad \text{boolean procedure}\ast \\
1011101 & \quad \text{real array} & 0100001 & \quad \text{procedure}\ast \\
1011110 & \quad \text{integer array} & 1010001 & \quad \text{switch} \\
1011011 & \quad \text{boolean array} & 0000001 & \quad \text{label} \\
& & 0000000 & \quad \text{string}
\end{align*}
\]

* Included here only for completeness.

3) Procedures which are parameters.

a) The entry for the formal procedure is

\[
Nttttttt
\]

where \( N \) is set to 1 if the formal procedure has parameters, and the seven bits \((t)\) represent the type of the formal procedure according to the scheme in (2b) above (marked with an asterisk).

b) Parameters to the formal procedure are entered as in (2) above.

c) A parameter to the formal procedure which is itself a procedure is entered as the single entry

\[
\emptysetttttt
\]

as in (3a) above, with \( N=\emptyset \).

4) The marker SE (Specification End) \((X'FF')\) is used to terminate the entries for a normal or a formal procedure (but not in case (3c) above).

5) The information for the entries described in (3a), (3b), (3c) is obtained from the comment specification for the formal procedure.
2. **BRIEF ILLUSTRATION OF THE INTERMEDIATE CODE IN USE**

The Intermediate Code may be understood as an object language which on execution operates on a simple push-down operand stack. Before proceeding, it is important to note that throughout this section, references to stacking in fact refer to a notional stack imagined to exist for the Intermediate Code to operate on in a straightforward non-optimal way. These references to stacking are not necessarily to be applied to the stack used by the machine code object programs actually generated by this compiler. Whenever possible, the stack visualised for the Intermediate Code to operate on will be referred to as the notional stack.

A simple arithmetic statement will be used as an illustration.

Presuming A, B, C and D to have been declared as 'REAL',

\[
A := B + C \times D
\]

will have the Intermediate Code representation (reverse Polish):

| TRA (k,A) | Place address of A on notional stack |
| TRR (k,B) | Place value of B on notional stack |
| TRR (k,C) | Place value of C on notional stack |
| TRR (k,D) | Place value of D on notional stack |
| *         | Multiply top two items of notional stack, leaving 1 result |
| +         | Add top two items of notional stack, leaving 1 result |
| ST        | Store result into A, leaving the notional stack empty. |

TRA stands for Take Real Address, TRR stands for Take Real Result, ST stands for Store. All Take operations place an item on the notional stack. All dyadic operations such as multiply (*) and add (+) operate on the top two items of the notional stack, replacing them by a single result item. The operation store (ST) requires the top two items of the notional stack to be a result and an address, with the result item uppermost; its action is to store the result item into the location specified by the address item, finally deleting both the result item and the address item from the stack.

These manipulations all refer to the working area of the notional stack. The quantities (k,A), (k,B), etc., which define A, B, etc., are themselves addresses in the notional stack, but they are not addresses in the working area. The Intermediate Code operations generated at the declaration or specification of the quantities A, B, etc., define locations on top of the existing notional stack specifically to hold the values associated with A, B, etc., and the corresponding addresses (k,A), (k,B), etc., are thereafter reserved for this use alone. This applies throughout the current block level (but not necessarily throughout the current hierarchy k). Having reserved all such locations on the notional stack for the declared and specified variables, the remainder of the notional stack is used as a working area, as indicated in the example above.

Briefly, every time a new hierarchy is entered, a region of the notional stack is reserved to hold all the parameters to the hierarchy (procedure)
and all the information defining the variables declared in all contained blocks. This is called the first order working storage. The remainder of the notional stack is then used as a working region, and for holding any array elements. On exit from the current hierarchy (procedure), the whole region of the notional stack associated with it is thrown away. Note that the reservation of space on the notional stack for all variables declared in contained blocks is done in an optimal way, such that blocks at the same level in the hierarchy share the same or overlapping areas for their declared variables.

To terminate this brief introductory section, an Algol program will be presented, consisting of the arithmetic statement used above but this time in a procedural environment, together with an annotated listing of the resulting Intermediate Code.

```
'BEGIN' 'REAL' A,B,C,D; 
   'PROCEDURE' P(X); 'VALUE' X; 'REAL' X; 
   A:= X + C * D; 
   P(B) 
'END' 

The generated Intermediate Code is

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEG (1)</td>
<td>Start block level 1</td>
</tr>
<tr>
<td>PJ (G0)</td>
<td>Jump to by-pass the procedure declaration</td>
</tr>
<tr>
<td>F0 : PE (1,12),12,1</td>
<td>Start hierarchy k=1, reserving 12 words on the notional stack for procedure P's link data and its parameter X.</td>
</tr>
<tr>
<td>CP</td>
<td></td>
</tr>
<tr>
<td>TRA (0,10)</td>
<td>A</td>
</tr>
<tr>
<td>TRR (1,10)</td>
<td>X</td>
</tr>
<tr>
<td>TRR (0,14)</td>
<td>C</td>
</tr>
<tr>
<td>TRR (0,16)</td>
<td>D</td>
</tr>
<tr>
<td>*</td>
<td>+</td>
</tr>
<tr>
<td>ST</td>
<td>RETURN</td>
</tr>
<tr>
<td>G0 : TRA (0,10)</td>
<td>A</td>
</tr>
<tr>
<td>TRR (0,12)</td>
<td>B</td>
</tr>
<tr>
<td>TRR (0,14)</td>
<td>C</td>
</tr>
<tr>
<td>TRR (0,16)</td>
<td>D</td>
</tr>
<tr>
<td>TIC (4)</td>
<td>Put integer 4 on the notional stack</td>
</tr>
<tr>
<td>STA }</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>Store 4 into D,C,B,A successively (the operation STA leaves 4 on the stack but deletes the destination address).</td>
</tr>
<tr>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>CF (1,0),00</td>
<td>Initiates the call of procedure F0 (&quot;Call Function&quot;)</td>
</tr>
<tr>
<td>PDR (G1)</td>
<td></td>
</tr>
<tr>
<td>TRR (0,12)</td>
<td>B</td>
</tr>
<tr>
<td>EIS</td>
<td></td>
</tr>
<tr>
<td>G1 : APP</td>
<td>Marker for end of parameters</td>
</tr>
<tr>
<td>BEND (1)</td>
<td>End block level 1 (that is main program)</td>
</tr>
<tr>
<td>FINISH</td>
<td></td>
</tr>
</tbody>
</table>
```

End of the procedure. Uses links in notional stack to return to point of call.

Initiates the call of procedure F0 ("Call Function")

Marker for end of parameters

End block level 1 (that is main program)
2A. Note on the Translation of Designational Expressions

The following conventions are used when translating a designational expression into Intermediate Code:

(i) Within an actual parameter expression or a switch declaration:
The operations TL, TFLW, TFLV, and INDR (when used in a switch call), have the run-time effect of planting the resulting label onto the notional stack. No transfer of control is effected. A stacked label is a two-word item (8 bytes) containing

Topmost word:- address of start of working area in notional stack for the level at which the label is declared

Second word:-
   a) \( N \) (1 byte), the block level within the hierarchy at which the label was declared;
   b) address in the compiled program pointed at by the label.

(ii) Elsewhere:
   a) The operation TL at run time, having evaluated the label, effects an immediate transfer of control. Nothing is stacked.
   b) The operations TFLW, TFLV, and INDR (when used in a switch call), at run time, all operate on the previously stacked label (see (i) above), effecting an immediate transfer of control.

Using these conventions, there is no need for the compiler to generate explicit transfer-of-control instructions, such as the GTA and GTFA of Whetstone KDF9 Algol.
3. **BLOCKS**

Within a hierarchy, block structure is represented in the Intermediate Code by:

\[
\begin{align*}
\text{BE} (n) & \quad (\text{BE} = \text{Block Entry}) \\
. & \\
. & \\
. & \\
. & \\
\text{BEND} (n) & \quad (\text{BEND} = \text{Block End})
\end{align*}
\]

\(n\) is the block nesting number in the current hierarchy, starting from one at the start of the procedure body. For a labelled procedure body, \(n\) for the label is one, while \(n\) for the start of the procedure body becomes two.

An Intermediate code block is created only at a 'BEGIN' followed by declarations, terminating at the corresponding 'END'. Space on the notional stack, to hold the necessary information about the declared quantities, is available throughout the period in which the block is active. This information is held in the first order working storage (see Section 2). Variables declared as integer or boolean occupy one word (4 bytes), variables declared as real occupy two words (8 bytes) and must be double-word aligned by the compiler, and quantities declared as arrays occupy four words (16 bytes).
A. PROCEDURES

4A. Procedure Structure in the Intermediate Code

Declarations

The form of the Intermediate Code generated for a procedure declaration is illustrated in the following schematic:

```
UJ(Gg)                  Present to prevent execution of the procedure before it is called.
Pp : PE(k,L),m,N       Procedure Entry operation.
  Parameter list       To set up the parameters ready for operations
                        use in the procedure body, if necessary.
                        See section AE.
Procedure body         Normal Intermediate Code operations.
RETURN                 Returns to the point of call of this procedure.
Gg:                    If several procedure declarations
                        are presented together, label Gg
                        is not assigned until the last
                        such procedure has been translated
                        into Intermediate Code. That is,
                        more than one procedure declaration
                        may be by-passed by one UJ operation.
```

Note that at a procedure declaration, the operation PE is always assigned a P label, which is used both in the procedure call mechanism and for diagnostic purposes.

Calls

The form of the Intermediate Code generated at a procedure call is illustrated in the following schematic:

```
Call operation         See section AC for details of call operations
Actual operations      Define the actual parameters of this procedure. See section AD.
APP                     Marks the end of the list of actual operations.
```

4B. Parameter List Operations

The full list of these operations is:

```
CRFA                   Copy real formal array
CIFA                   Copy integer formal array
ABFA                   Copy boolean formal array
CA                     See below
CP                     See below

These intermediate code operations are all parameterless. The operations CA, CP are used only to locate the array word referred to by a subsequent copy operation (arrays by value).
```
Calls of Function Designators and Procedure Statements

The call operations for the various kinds of procedure (or function) are:

| CF (k,p),π | Call function                  | CFZ(k,p), π | Call function zero |
| CTF(k,p),π | Call type function             | CTFZ(k,p), π | Call type function zero |
| CFF(k,s),π | Call formal function           | CFFZ(k,s),π | Call formal function zero |
| CTFF(k,s),π | Call type formal function      | CTFFZ(k,s),π | Call type formal function zero |

The operations in the righthand column are used to call procedures with no parameters, and are distinguished by the addition of the descriptor 'zero'.

k is the hierarchy number of the called procedure, p is the index number of the called procedure, π is the pointer to level parameters (b). The combination (k,s) for a formal procedure is the stack address of the formal parameter location for that procedure.

A function call which invokes a procedure as a function designator uses one of the operations CTF, CTFZ, CFF, CFFZ.

The corresponding call invoking a procedure as a procedure statement uses one of the operations CF, CFZ, CFF, CFTZ.

A location on top of the notional stack is reserved for the result of a type procedure. Use of a procedure statement will result in the deletion of the unused or superfluous result location.
**Actual Operations**

At a procedure call, any actual parameters are translated into Intermediate Code involving some of the following special-purpose operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Parameters</th>
<th>Intermediate Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR(Gg)</td>
<td>Parameter subroutine</td>
<td></td>
<td>EIS</td>
<td>End implicit subroutine</td>
</tr>
<tr>
<td>PRA(k,s)</td>
<td>Par. real array</td>
<td>k, s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLA(k,s)</td>
<td>Par. integer array</td>
<td>k, s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBA(k,s)</td>
<td>Par. boolean array</td>
<td>k, s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSW(k,1)</td>
<td>Par. switch</td>
<td>k, s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPR(k,p)</td>
<td>Par. procedure</td>
<td>k, p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIF(k,p)</td>
<td>Par. integer function</td>
<td>k, p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIF(k,p)</td>
<td>Par. real function</td>
<td>k, p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBF(k,p)</td>
<td>Par. boolean function</td>
<td>k, p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PST ('STRING')</td>
<td>Par. string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEST</td>
<td>Par. end string</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every actual parameter is translated into a sequence of Intermediate Code operations. This sequence of intermediate code operations is called an implicit (parameter) subroutine. The first operation in an implicit subroutine is always PSR, the last operation is always EIS. The label parameter Gg to PSR points to the next PSR or to APP, whichever is appropriate.

When an actual parameter is a (formal) array, a (formal) switch, a (formal) procedure, or a (formal) string, the body of the implicit subroutine is the appropriate single parameter operation from the above list. Formal array parameters use the operations PRA, PLA or PBA.

When an actual parameter is a constant, the body of the implicit subroutine is one of the operations TIC, TRC, TBC.

When an actual parameter is a string, the body of the implicit subroutine consists of the operation PST, followed by the string in Algol basic symbols (including the opening and closing string quotes), followed by the marker operation PEST.

When an actual parameter is an expression or a designational expression, the body of the implicit subroutine is the sequence of Intermediate Code operations (not parameter operations) necessary to evaluate the appropriate result. This includes label identifiers.

When an actual parameter could be a valid left hand side of an assignment statement then the body of the implicit subroutine is the sequence of Intermediate Code operations (not parameter operations) necessary to evaluate the appropriate address.
Example of a Procedure Call

The following rather unlikely procedure statement will be translated into Intermediate Code as an illustration of the generation of actual parameters.

\[ P(A, A, 'TRUE', X + Y, C, 'IF' I = \emptyset 'THEN' R1 'ELSE' R2, \]

\[ S, Q, '(PING)', P, B'<1, 4>', T'<2'>; \]

The following types are assumed for the variables occurring in this procedure statement:

- \( A \) real
- \( X \) real
- \( Y \) real formal
- \( C \) real
- \( C \) real array
- \( I \) integer
- \( R1 \) label
- \( R2 \) label
- \( S \) switch
- \( Q \) integer procedure
- \( F \) procedure formal
- \( B \) integer array
- \( T \) switch

All parameters will be written symbolically, e.g. no numbers will be supplied for hierarchy numbers, block levels or stack addresses. Label numbers will be assumed to start from zero.

The Intermediate Code translation is:

\[ \text{CF}(k, p), T \]

\[ \text{PSR}(G\emptyset) \]

\[ \text{TRA}(k, A) \]

\[ \text{EIS} \]

\[ G\emptyset: \text{PSR}(G1) \]

\[ \text{TIC}(A) \]

\[ \text{EIS} \]

\[ G1: \text{PSR}(G2) \]

\[ \text{TBCT} \]

\[ 'TRUE' \]

\[ \text{EIS} \]

\[ G2: \text{PSR}(G3) \]

\[ \text{TAR}(k, X) \]

\[ \text{TFR}(k, Y) \]

\[ + \]

\[ \text{EIS} \]

\[ X+Y \]

\[ G3: \text{PSR}(G4) \]

\[ \text{FRA}(k, C), - \]

\[ \text{EIS} \]

\( C \)
G4: PSR(G5)
  BEX
  TIR(k,1)
  TIC®
  =
  IFJ(G6)
  TL(k,R1),n
  UJ(G7)
G6: TL(k,R2),n
CEND
G7: EIS
G5: PSR(G8)
  PSW(k,S),-
  EIS
G8: PSR(G9)
  PIF(k,Q),π
  EIS
G9: PSR(G10)
  PST('PING')!
  PEST
  EIS
G10: PSR(G11)
  PPFR(k,F),π
  EIS
G11: PSR(G12)
  TLLA(k,B),2
  TIC1
  TIC(A)
  INDR(2)
  EIS
G12: PSR(G13)
  SAPP(k,T)
  TIC(2)
  INDR(1)
  EIS
G13: APP
AF. Use of Formal Parameters within a Procedure Body

The operations used within a procedure body to refer to a formal parameter fall into the following classes, depending on the specification of the formal parameter:

a) By value

On entry to a procedure, the parameter locations set up for parameters called by value are initialised with the actual values of these parameters at the point of call of the procedure. By this means they are made local to the procedure body, and are operated on by Intermediate Code operations of the normal (non-formal) kind such as TIR, TRA (i.e. Intermediate Code operations whose mnemonic does not contain F for 'formal'). The stack parameters (k, s) to these Intermediate Code operations are the stack addresses of the parameter locations. This paragraph does not apply to labels by value (see below, (c)).

b) By name

The following Intermediate Code operations are used, in which (k, s) denotes the stack address of the appropriate formal parameter location:

\[
\begin{align*}
TFRA (k, s) & \quad \text{Addresses of arithmetic or boolean formal variables (scalars)} \\
TFLA (k, s) & \\
TFBA (k, s) & \\
TFR (k, s) & \quad \text{Values of arithmetic or boolean formal variables (scalars)} \\
TFI (k, s) & \\
TFB (k, s) & 
\end{align*}
\]

c) Switches

For a formal switch, the operation TFS (k, s) is used instead of the operation SAPP (k, l). See Section 5.

d) Procedures

The operations used are

\[
\begin{align*}
CFF (k, s), \pi & \quad \text{Call formal function} \\
CTFF (k, s), \pi & \quad \text{Call type formal function} \\
CFFZ (k, s), \pi & \quad \text{Call formal function zero} \\
CTFFZ (k, s), \pi & \quad \text{Call type formal function zero}
\end{align*}
\]

See Section (AC).
e) **Labels**

The operations used are

\[ TFLN(k,s) \] Take formal label by name.
\[ TFLV(k,s) \] Take formal label by value.

f) **Arrays**

The parameter locations on the notional stack for formal array parameters by name or by value contain exactly the same information as is used elsewhere for arrays. Use of such parameters therefore does not involve special operations. To be specific, the only operations involved for any array accessing whatever are:

\[ TBA(k,s),b \] Take boolean array address
\[ TIAA(k,s),b \] Take integer array address
\[ TRLA(k,s),b \] Take real array address

and the operations INDA(b) and INDR(b).
5. **SWITCHES**

5A. **Switch Structure in the Intermediate Code**

**Declarations**

The form of the Intermediate Code generated for a switch declaration is illustrated in the following schematic:

- \( UJ(G\phi) \) Present to prevent execution of the switch before it is called.
- \( L1: DSI(G1) \) Decrement switch index (see below)
  - Evaluate 1st switch element
  - \( EIS \) End implicit subroutine (terminator for each element)
- \( G1: DSI(G2) \) Evaluate 2nd switch element
  - \( EIS \)
  - ...
- \( G(n-1): DSI(Gn) \) Evaluate nth switch element
  - \( EIS \)
- \( Gn: ESL \) End switch list (see below)
- \( G\phi : \)
  - ...
  - ...
  - ...
  - ...

A switch is called by a switch designator with a subscript expression which is evaluated at the point of call. On transfer of control to the declared switch at the label \( L1 \), the function of the operation \( DSI \) is to subtract one from the value of the subscript expression and to jump to the next \( DSI \) if the result is non-zero. If this process continues until the operation \( ESL \) is encountered, a failure condition exists, because the specified subscript is out of the range defined by the switch list. Having found a \( DSI \) which reduces the subscript to zero (the \( m \)th \( DSI \) if the original value of the subscript was \( n \)), the appropriate switch element is evaluated, and the associated \( EIS \) causes a return to the point of call of the switch.

Note that a switch declaration is not made into a block. Note also that the first \( DSI \) in a switch declaration is assigned an \( L \) label, which is used both in the switch designator mechanism and for diagnostic purposes.
Designators

The form of the Intermediate Code generated for a switch designator is illustrated in the following schematic:

- \text{SAPP}(k,1) \quad \text{Switch approaching}
- \text{INDA}(1) \quad \text{Index address}

The operation \text{INDA} is also used when evaluating subscripted variables, but the context is always sufficient to determine which use is intended.
5B. **Switch Designators and Array Elements as Actual Parameters**

If an actual parameter in a procedure call is written as e.g.,

\[ S!\langle 2\rangle \]

it may not be known whether \( S \) is a switch or an array until later in the translation. It is therefore necessary to allow space for either eventuality in the Intermediate Code.

The alternative final results are:

<table>
<thead>
<tr>
<th>Switch designator</th>
<th>Array element</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAPP((k,1))</td>
<td>corresponds to TRAA((kms),1)</td>
</tr>
<tr>
<td>DUMMY</td>
<td></td>
</tr>
<tr>
<td>INDA((1))</td>
<td>corresponds to INDA((1))</td>
</tr>
<tr>
<td>TFS((k,s))</td>
<td>corresponds to TRAA((k,s),1)</td>
</tr>
<tr>
<td>DUMMY</td>
<td></td>
</tr>
</tbody>
</table>

**DUMMY** is a one byte Intermediate Code operation whose only function is to fill space.

For the detailed implementation of this technique, refer to the documentation of Phase 1\(\phi\).
6. **CONDITIONAL STATEMENTS AND EXPRESSIONS**

The form of the Intermediate Code generated for a conditional statement or expression will be illustrated schematically for the case 'IF' B 'THEN' C 'ELSE' D, where C and D are either both expressions or both statements:

- **BEX**
  - Translation of B
  - IFJ(Gφ)
- **Boolean expression marker**
  - If false jump (i.e. if B false)
- **Translation of C**
  - UJ(G1)
- **Unconditional jump**
- **Gφ: Translation of D**
  - GEHD
- **Conditional end marker**

This implementation permits D itself to be a conditional statement or expression. In the case of a conditional statement 'IF' B 'THEN' C, where C is a statement, the translation into Intermediate Code is:

- **BEX**
  - Translation of B
  - IFJ(Gφ)
  - Translation of C
  - GEHD
- **Gφ:***
  - ...
  - ...
  - ...

---

*ENGLISH ELECTRIC LEO MARCONI*
7. ARRAYS

Declarations

The form of the Intermediate Code generated for an array declaration

\[
\{ ('\text{REAL}') \}
\{ ('\text{INTEGER}') \}
\{ ('\text{BOOLEAN}') \}
\text{ARRAY} \; A, B, \ldots, K \langle L1: L2, M1: M2, \ldots, Z1: Z2 \rangle
\]

where A, B, ..., K represent the array identifiers, and L1 and L2, M1 and M2, ..., Z1 and Z2 represent expressions defining the lower and upper bounds respectively of the first, second, ..., last dimensions, is illustrated in the following schematic:

Translation of L1
STACK

Translation of L2
STACK

Translation of M1
STACK

...

Translation of Z1
STACK

Translation of Z2
STACK

Make Storage Function
See below for the forms of the operation.

The possible forms of the Make Storage Function operation are:

\[
\text{MBSF}(k, s), a
\]

Make boolean storage function

\[
\text{MIF}(k, s), a
\]

Make integer storage function

\[
\text{MRSP}(k, s), a
\]

Make real storage function

where \((k, s)\) is the stack address of the last of the stacked array items for A, B, ..., K, from which the rest may be found. The stacked array items, each containing all the defining information for each individual array, are normally referred to as the array words, although in fact they each occupy four words on System 4 (16 bytes). All these array words generated from one array segment share the same storage function.

\(a\) is the number of arrays in the array segment (i.e. the number of the arrays A, B, ..., K). This is also the number of array words generated by the make storage function operation.
**Subscripted Variables**

The form of the Intermediate Code generated for a subscripted variable $A[I,J,\ldots,Z']$, where $I,J,\ldots,Z$ are arithmetic expressions each of which may be a further subscripted variable, is illustrated by the following schematic:

- **Take array address**
- **Evaluate I**
- **Evaluate J**
- **Evaluate Z**
- **INDR(b) or INDA(b)**

See below for possible forms of this operation.

Index result or Index address, according to context.

The possible forms of the Take array address operation are:

- **TBA(k,s),b** Take boolean array address
- **TIA(k,s),b** Take integer array address
- **TRA(k,s),b** Take real array address

where $(k,s)$ is the stack address of the corresponding array word, and $b$ is the number of dimensions of the array.

The operation INDR(b) delivers the value of the subscripted variable, INDA(b) delivers its address (on the notional stack).
### 8. FOR STATEMENTS

#### 8A. General

A for statement is treated as a block which opens with the Intermediate Code operation:

**FBB(n)**  
For block entry,

which performs exactly the same functions as the operation **B6(n)**, and terminates with the Intermediate Code operation

**BND(n)**  
Block end

as used for normal blocks (see Section 3).

Because a for statement requires three parameters to control the running of the various for list elements, these three quantities are 'declared' at this point, that is, space is reserved for them, as if they had been declared explicitly, in the first order working storage (see Sections 2,3). The three parameters are:

<table>
<thead>
<tr>
<th>Stack address (words)</th>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k,s)</td>
<td>SA(boolean)</td>
<td>Marker controlling the action of the Intermediate Code operation AST (see Section 8E)</td>
</tr>
<tr>
<td>(k,s+1)</td>
<td>FI(integer)</td>
<td>The for index (see below and Section 8E). Holds the initial value of the controlled variable each time round a for list element. Phase 10 treats S3 as real, but Phase 20 later determines its type to be the same as that of the controlled variable.</td>
</tr>
<tr>
<td>(k,s+2)</td>
<td>S3(real)</td>
<td></td>
</tr>
</tbody>
</table>

The stack address (k,s) is double-word aligned.

An extra block is created if the controlled statement is itself a block or a labelled block.

The quantity TI (Test Index) is also used but does not reside on the stack. At the beginning of each for list element it is a copy of FI. TI is the item actually decremented (in a register) by the operation DFI.
Several Intermediate Code operations have been introduced specifically for use in the translation of for statements. These operations are:

- DFI(Gg)  Decrement for index
- AST(Gg)  Avoid step
- STN      Special purpose ST operation with run time effect of ST
- STPA     Store from address - special purpose ST operation
- FMIT(k,SA)  For initialise
- CSM      Controlled statement marker
- STUN(Gg)  Step-until macro operation
- FI-1     Decrement FI by 1, used in while elements
- STEP     Marker used in step until elements

The use of these operations will be clear from the examples which follow.
For Statement Structure in the Intermediate Code

The form of the Intermediate Code generated for a for statement of the form

'FOR'V:= first element, second element,..., last element 'DO'
Controlled statement, is illustrated in the following schematic:-

\[
\begin{align*}
\text{PBE(}n) & \quad \text{For block entry} \\
\text{FNTT(}k, S_{24}) & \quad \text{Initialise by setting } S_A \\
G_0 : \text{Stack} & \Rightarrow \text{address of } V \\
S_3 & \Rightarrow \text{value of } V \\
F_1 & \Rightarrow F_1 + 1 \\
DFI(G_1) & \quad \text{Translation of first element} \\
UJ(G_0) & \quad \text{Return for next element} \\
G_1 : DFI(G_2) & \quad \text{Translation of second element} \\
UJ(G_0) & \quad \text{Return for next element} \\
G_2 : & \quad \text{...} \\
\vdots & \quad \text{...} \\
G_{(m-1)} : DFI(G_m) & \quad \text{Translation of last (mth) element} \\
CSE & \quad \text{End of the for statement} \\
G_c : \text{Controlled statement} & \quad \text{return for next element} \\
UJ(G_0) & \quad \text{End of the for statement} \\
G_m : BEND(n) & \quad \text{End of the for statement}
\end{align*}
\]

The formats of the various kinds of for list element will now be described.
8C. Arithmetic Element

An arithmetic for list element is an arithmetic expression, $A$, say. The Intermediate Code form of $A$ as an arithmetic for list element is illustrated in the following schematic:

```
DUMMY
DUMMY
DUMMY
Translation of A
STN
```

Stores value of $A$ into the location $V$ whose address has been pre-stacked (see 8B).

This coding fits into the schematic in (6B) where the words "Translation of (nth) element" appear.

The three DUMMY operations are generated in case this element turns out to be a step-until element (see (6E)).

The special operation STN is generated instead of ST as an aid to the subsequent processing of the Intermediate Code by Phase 20. It is effectively a directive to Phase 20 to remember the address of $V$ for use in translating subsequent elements into machine code. The machine code translation of STN is identical to that of the normal operation ST.
6D. While Element

The form of the Intermediate Code for a while element of the form C\WHILE\B, where C is an arithmetic expression and B is a boolean expression, is illustrated in the following schematic:

DUM Y
DUM Y
DUM Y
Translation of C
STN
Stores value of C into V, whose address has been pre-stacked.

BEX
Translation of B
IFJ(GΦ)
Return for next element if current element exhausted.

FI-1
Performs the function FI:=FI-1

This coding fits into the schematic in (6B) where the words "Translation of (nth) element" appear.

The three DUM Ys and the STN operation are present for the reasons listed in (6C).

It may be of interest to note that in this occurrence of IFJ(GΦ), G0 is a backward reference. With this unique exception, the label parameter to IFJ is always a forward reference.
8E. *Step Until Element*

The following schematic illustrates the form of the Intermediate Code generated for a step until element of the form

\[ \text{DISTEP'E'UNTIL'F}, \]

where D, E, and F are arithmetic expressions:

- \( \text{AST}(Gg) \)
  - Translation of D
  - Avoid step
- STEP
- \( Gg: \)
  - Translation of E
  - Marker to terminate D
- Translation of F
- \( \text{STUN}(G\emptyset) \)
  - Step until macro

This coding fits into the schematic in (8B) where the words "Translation of (nth) element" appear.

\( \text{AST} \) \( (Gg) \) takes the place of the three DUMAYs generated for arithmetic and for while elements. The function of this operation is to jump to \( Gg \) if \( S_A \) is false.

The function of \( \text{STUN} \) \( (G\emptyset) \) is to perform the step using \( S_3 \) and \( E \) to evaluate the new value of \( V \), to set \( S_A \) correctly, and to exit to \( G\emptyset \) when the element is exhausted, otherwise to perform \( FI:=FI-1 \) and to pass on to the controlled statement. At run time, \( \text{STUN} \) becomes a call of a Slave Routine (see Section (Aa(b))).
SF. Example of a For Statement

The Intermediate Code translation of the following for statement will be presented:

```
'FOR' A := 1 'STEP' 2 'UNTIL' 15, A+2 'WHILE' A'LT' 15
'DO' I := A;
```

A is assumed to be 'REAL', I to be 'INTEGER'. Locations for S4 (boolean), FI (integer) and S3 (real) are also created. The translation is:

```
FBE(n)
FNIT(k, S4)
G∅: TRA(k, A)
     TTL(k, S3)
STEA
     Address of A on the notional stack

TIA(k, FI)
TIR(k, FI)
TIC1
     +
STN
     Performs FI := FI + 1

DFI(G1)
AST(G2)
TIC1
     STEP
G2:
     TIC(2)
     TIC(∅)
     STUN(G∅)
     UJ(G3)
G1: DFI(0A)
     DUMMY
     DUMMY
     TIC(15)
     STN
     Start of second element, 15.
```

Transfer control to controlled statement

Start of first element, 1 'STEP' 2 'UNTIL' 15

Perform A := 15
| **G4:** DFI(G5) | Transfer control to controlled statement |
| **DUUHY** | Start of third element, A+2 WHILE A'LT'1Ø |
| **DUEHY** | |
| **TRR(k,A)** | |
| **TIC(2)** | |
| **+** | |
| **STU** | |
| **BEX** | |
| **TRR(k,A)** | |
| **TIC(1Ø)** | |
| **<** | |
| **IFJ(GØ)** | |
| **FI-1** | |
| **CSM** | |
| **G3:** TLA(k,1) | |
| **TRR(k,A)** | |
| **ST** | |
| **UJ(GØ)** | |
| **G5:** END(n) | |

**Perform:** A:=A+2

**Boolean expression marker**

**Return for next element if this element is exhausted.**

**Perform:** FI:=FI-1

**Controlled statement marker**

**Perform controlled statement I:=A,**

**Return for next element.**

**End of for statement**
9. **OWN VARIABLES**

Own variables in the Intermediate Code are identified as hierarchy \( k = -1 \). That is, for an own variable \( k \) is 4 bits of all ones. Because of this convention, there are no Intermediate Code operations referring explicitly to own variables.

Own variables at run time do not reside on the notional stack, but in a special area which also holds the constants required by the generated program. Space for all own variables used in a program is reserved immediately on entry to a running program.
10. **NOTE ON INDEPENDENT COMPILATION OF ALGOL PROCEDURES**

Algol procedures may be presented to the System A Algol Compiler for independent compilation, provided they are presented to the compiler in the form

```
BEGIN
<procedure declaration>
END
```

(The brackets<> are not literals but are used in the Backus normal form sense. See <procedure declaration> in paragraph 5.4.1 of the Revised Report on Algol 60.) The result of such a compilation is a machine code module which may be called from any other Algol module or base program. The Intermediate Code form of an independently compiled procedure, as would be expected from the form of the Algol above and from Section (44), is:

```
XX(1)
UJ(C®)
PO: FE(1,L),N,H
Parameter list operations
Procedure body
RETURN
C®: BEND(1)
```

In a module or base program which calls an independently compiled procedure, the source Algol must contain a specification of the called procedure. The syntax of this specification, in the notation of paragraph 5.4.1 of the Revised Report on Algol 60, is

```
<specification of independently compiled procedure>:=
   'PROCEDURE' <procedure heading> 'ENTER' <module entry name>
   <type>'PROCEDURE' <procedure heading> 'ENTER' <module entry name>
```

Semantically,

```
'ENTER' <module entry name>
```

stands in place of

```
<procedure body>
```

in the definition of a procedure declaration. The module entry name is the entry name assigned to the independently compiled procedure when it was compiled. This specification must be identical with the specification part of the independently compiled procedure. It is the user's responsibility to ensure that this is so, since the compiler makes no check (except that the amount of space reserved for parameters is checked at run time).
In conformity with Section (4.4), the Intermediate Code form of the specification of an independently compiled procedure is

\[
\text{UJ}(Gg) \\
Pp: \text{PL}(k,l), m, n \\
\text{Parameter list operations} \\
\text{ENTER (module entry name)} \\
\text{RETURN} \\
Gg: \cdot \\
\cdot \\
\cdot
\]

ENTER is here an Intermediate Code operation whose parameter is the entry name of the called module, up to eight characters in length. This method of implementation incidentally means that the procedure identifier used in the calling module need not be the same as the procedure identifier of the called module.

Example:

Calling module

\[
\text{'BEGIN' 'INTEGER' } X; \\
\text{'REAL' 'PROCEDURE' FORGE(A,B,C); } \\
\text{'INTEGER' A,B,C; } \\
\text{'ENTER' MODULE1; } \\
X:= \text{FORGE}(4,3,2) \\
\text{'END'}
\]

Called module

\[
\text{'BEGIN' 'REAL' 'PROCEDURE' ANVIL(U,V,W); } \\
\text{'INTEGER' U,V,W; } \\
\text{ANVIL:= WIU(V,W); } \\
\text{'END'}
\]

Compiled with entry name MODULE1.

The call of the procedure FORGE in the calling module is in fact a call of the independently compiled procedure ANVIL, whose entry name is MODULE1.
11. **LIBRARY PROCEDURES**

All standard input-output procedures, 42 in number, may be used without the specifications described in Section(10), as are all 9 standard functions ABS, SIGN, SQRT, SIN, COS, ARCTAN, LN, EXP and ENTER.

The full list of the 42 standard input-output procedures appears in the Algol Reference Manual. If these procedures are used without specification, then the compiler will automatically supply specifications of the standard library versions. If however the user does not desire to use the standard library version of one of these procedures, then he must supply the Algol specification giving the entry name of the module he wishes to use instead, as in Section 10.

The Intermediate Code specifications of the standard procedures supplied by the compiler appear as the final items in the generated Intermediate Code, and they have the format:

```
PE(1,L), m, N
ENTER (module entry name)
RETURN
```

In the special case of the four procedures FORMAT, ABS, SIGN, and ENTER, the Intermediate Code specifications generated by the compiler use the parameterless Intermediate Code operations FORMAT, ABS, SIGN, and ENTER in place of ENTER (module entry name). This is because the four named procedures are implemented without recourse to independently compiled modules. The user, of course, may still pre-empt this system by supplying a specification of a module he prefers to the standard versions that would otherwise be built into his object program.
12. THE INTERMEDIATE CODE OPERATION TRACE

If the option ROUTE is specified on the // COMPILE card for an Algol module (see PS A.13.5 (5J/7400) and Algol Reference Manual Part 2), then the Intermediate Code operation TRACE (8 character identifier in 6 bytes in packed ABS form) is generated in the following positions:

a) Labels

At every use of a label on the left-hand side (a "declaration") the operation TRACE is generated with the identifier of the label itself as parameter.

* e.g. L1: FLU2: 'END'

would be translated into

L1: TRACE (L1)
L(1+1): 'TRACE (FIN2)
BEND (n)

in which e.g. the parameter L1 appears in packed Algol basic symbol-form as 5C1.

b) Procedures

At every procedure declaration the operation TRACE is generated with the identifier of the procedure itself as parameter.

* e.g. 'PROCEDURE STRUH(X);...

would be translated into

Pp : PE (k, L), n, N
TRACE (STRUH)

This operation TRACE permits an explicit run-time listing of labels passed and procedures entered to be made.
13. **THE INTERMEDIATE CODE OPERATIONS FAIL AND LFAIL**

When Phase \( \Phi \) detects an error condition at the point of error in an Algol source program, e.g. a mis-spelt basic symbol, then the Intermediate Code operation

\[ \text{FAIL}(r) \]

is generated, where \( r \) is the serial number of the current failure (i.e. the \( r \)th failure detected so far). Phase \( \Phi \) then recovers and proceeds with the translation and generation of Intermediate Code. This type of fail operation is called an immediate failure.

When Phase \( \Phi \) detects a failure condition later than the actual point of error (e.g. an identifier not declared), then the Intermediate Code operation

\[ \text{LFAIL}(r, \text{skeleton operation}) \]

is generated. LFAIL stands for late failure. \( r \) has the same meaning as above. 'Skeleton operation' is a one-byte item whose purpose is to give the successor pass, Phase \( 2\Phi \), information about the type of operation that was present before Phase \( 1\Phi \) found it necessary to back-track and plant LFAIL. The method of doing this, and the whole problem of skeleton chaining, is dealt with in the documentation of Phase \( 1\Phi \). The five possible skeleton operations are:

- **R0** Result operation (e.g. TRR)
- **A0** Address operation (e.g. TRA)
- **S0** Statement operation (e.g. P; - a parameterless procedure call)
- **F0** Function operation (e.g. C)
- **PO** Parameter operation (e.g. PRA)
### Dictionary of Intermediate Code Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Hexadecimal Operation Value</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Standard function ABS</td>
<td>C1</td>
<td>None</td>
</tr>
<tr>
<td>APP</td>
<td>Actual Parameters terminal marker</td>
<td>19</td>
<td>None</td>
</tr>
<tr>
<td>AST</td>
<td>Avoid Stop</td>
<td>0E</td>
<td>g</td>
</tr>
<tr>
<td>BE</td>
<td>Block Entry</td>
<td>0A</td>
<td>n</td>
</tr>
<tr>
<td>BEND</td>
<td>Block End</td>
<td>05</td>
<td>n</td>
</tr>
<tr>
<td>BEX</td>
<td>Boolean Expression marker</td>
<td>FC</td>
<td>None</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>ID</td>
<td>None</td>
</tr>
<tr>
<td>CBFA</td>
<td>Copy Boolean Formal Array</td>
<td>6B</td>
<td>None</td>
</tr>
<tr>
<td>CEND</td>
<td>Conditional End Marker</td>
<td>FD</td>
<td>None</td>
</tr>
<tr>
<td>CF</td>
<td>Call Function</td>
<td>86</td>
<td>(k, p), n</td>
</tr>
<tr>
<td>CFF</td>
<td>Call Formal Function</td>
<td>96</td>
<td>(k, s), n</td>
</tr>
<tr>
<td>CFFZ</td>
<td>Call Formal Function Zero</td>
<td>98</td>
<td>(k, s), n</td>
</tr>
<tr>
<td>CFZ</td>
<td>Call Function Zero</td>
<td>88</td>
<td>(k, p), n</td>
</tr>
<tr>
<td>CIFA</td>
<td>Copy Integer Formal Array</td>
<td>6A</td>
<td>None</td>
</tr>
<tr>
<td>CP</td>
<td></td>
<td>10</td>
<td>None</td>
</tr>
<tr>
<td>CRFA</td>
<td>Copy Real Formal Array</td>
<td>69</td>
<td>None</td>
</tr>
<tr>
<td>CSM</td>
<td>Controlled Statement Marker</td>
<td>FE</td>
<td>None</td>
</tr>
<tr>
<td>CTF</td>
<td>Call Type Function</td>
<td>81</td>
<td>(k, p), n</td>
</tr>
<tr>
<td>CTFF</td>
<td>Call Type Function Formal</td>
<td>91</td>
<td>(k, s), n</td>
</tr>
<tr>
<td>CTFFZ</td>
<td>Call Type Function Formal Zero</td>
<td>99</td>
<td>(k, s), n</td>
</tr>
<tr>
<td>CTFZ</td>
<td>Call Type Function Zero</td>
<td>89</td>
<td>(k, p), n</td>
</tr>
<tr>
<td>DFI</td>
<td>Decrement For Index</td>
<td>0E</td>
<td>g</td>
</tr>
<tr>
<td>DSI</td>
<td>Decrement Switch Index</td>
<td>08</td>
<td>g</td>
</tr>
<tr>
<td>DUMMY</td>
<td>Dummy operation</td>
<td>FT</td>
<td>None</td>
</tr>
<tr>
<td>EIS</td>
<td>End Implicit Subroutine marker</td>
<td>1F</td>
<td>None</td>
</tr>
<tr>
<td>ENTER</td>
<td>Enter independently compiled procedure</td>
<td>07</td>
<td>Entry Name</td>
</tr>
<tr>
<td>ENTER</td>
<td>Standard function Enter</td>
<td>09</td>
<td>None</td>
</tr>
<tr>
<td>ESL</td>
<td>End Switch List marker</td>
<td>09</td>
<td>None</td>
</tr>
<tr>
<td>FAIL</td>
<td>Compilation Failure</td>
<td>1F</td>
<td>r</td>
</tr>
<tr>
<td>Operation/Mnemonic</td>
<td>Hexadecimal Operation Value</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>FBE</td>
<td>$6</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>FI-1</td>
<td>CB</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>FINISH</td>
<td>$6</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>FNIT</td>
<td>CS</td>
<td>((k,s4))</td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>C5</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>IFJ</td>
<td>B2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>INDA</td>
<td>6C</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>INDR</td>
<td>7D</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>LFAIL</td>
<td>1B</td>
<td>r, t</td>
<td></td>
</tr>
<tr>
<td>MBF</td>
<td>63</td>
<td>((k,s),a)</td>
<td></td>
</tr>
<tr>
<td>-SF</td>
<td>62</td>
<td>((k,s),a)</td>
<td></td>
</tr>
<tr>
<td>MRF</td>
<td>61</td>
<td>((k,s),a)</td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>F1</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>PBA</td>
<td>CF</td>
<td>((k,s),,-)</td>
<td></td>
</tr>
<tr>
<td>PBF</td>
<td>D3</td>
<td>((k,p),\pound)</td>
<td></td>
</tr>
<tr>
<td>PE</td>
<td>$1</td>
<td>((k,L),m,N)</td>
<td></td>
</tr>
<tr>
<td>PEST</td>
<td>CC</td>
<td>((none),m,N)</td>
<td></td>
</tr>
<tr>
<td>PFF</td>
<td>DB</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PFI</td>
<td>DA</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PFFR</td>
<td>D9</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PFS</td>
<td>DE</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PFSW</td>
<td>DF</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>-PPR</td>
<td>D8</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td>CE</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PIF</td>
<td>D2</td>
<td>((k,p),\pound)</td>
<td></td>
</tr>
<tr>
<td>PPR</td>
<td>D6</td>
<td>((k,p),\pound)</td>
<td></td>
</tr>
<tr>
<td>PRA</td>
<td>CD</td>
<td>((k,s),\pound)</td>
<td></td>
</tr>
<tr>
<td>PRF</td>
<td>D1</td>
<td>((k,p),\pound)</td>
<td></td>
</tr>
<tr>
<td>PSR</td>
<td>D1</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>PST</td>
<td>D6</td>
<td>'string'</td>
<td></td>
</tr>
<tr>
<td>PSW</td>
<td>D7</td>
<td>((k,l),\pound)</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>$2</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Operation Mnemonic</td>
<td>Hexadecimal Operation Value</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>SAPP</td>
<td>18</td>
<td>(k,1)</td>
<td></td>
</tr>
<tr>
<td>SIGN</td>
<td>C2</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>F8</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>F9</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>STEP</td>
<td>FA</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>STPA</td>
<td>FB</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>STN</td>
<td>FA</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>STUN</td>
<td>FA</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>A3</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TBLA</td>
<td>53</td>
<td>(k,s),b</td>
<td></td>
</tr>
<tr>
<td>TBCF</td>
<td>3B</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TBCST</td>
<td>37</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TBR</td>
<td>23</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFB3</td>
<td>2B</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFBBA</td>
<td>2B</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFLI</td>
<td>2A</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFLIA</td>
<td>4A</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFLN</td>
<td>49</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFLV</td>
<td>48</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFR</td>
<td>29</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFRA</td>
<td>49</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TFS</td>
<td>78</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TIA</td>
<td>42</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>TLLA</td>
<td>52</td>
<td>(k,s),b</td>
<td></td>
</tr>
<tr>
<td>TIC</td>
<td>32</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>TIC0</td>
<td>36</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TIC1</td>
<td>3L</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TIR</td>
<td>22</td>
<td>(k,s)</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Mnemonic</td>
<td>Hexadecimal Operation Value</td>
<td>Parameters</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>TL</td>
<td>Take Label</td>
<td>( A\phi )</td>
<td>((k,1), n)</td>
</tr>
<tr>
<td>TRA</td>
<td>Take Real Address</td>
<td>( A1 )</td>
<td>((k,s))</td>
</tr>
<tr>
<td>TRAA</td>
<td>Take Real Array Address</td>
<td>( 51 )</td>
<td>((k,s), b)</td>
</tr>
<tr>
<td>TRACE</td>
<td>Trace</td>
<td>( 9A )</td>
<td>identifier</td>
</tr>
<tr>
<td>TRC</td>
<td>Take Real Constant</td>
<td>( 31 )</td>
<td>constant</td>
</tr>
<tr>
<td>TRR</td>
<td>Take Real Result</td>
<td>( 21 )</td>
<td>((k,s))</td>
</tr>
<tr>
<td>UJ</td>
<td>Unconditional Jump</td>
<td>( B1 )</td>
<td>( g )</td>
</tr>
<tr>
<td>PLUS</td>
<td></td>
<td>( E\phi )</td>
<td>None</td>
</tr>
<tr>
<td>MINUS</td>
<td></td>
<td>( E1 )</td>
<td>None</td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td>( E2 )</td>
<td>None</td>
</tr>
<tr>
<td>DVDE</td>
<td></td>
<td>( E3 )</td>
<td>None</td>
</tr>
<tr>
<td>IDIV</td>
<td></td>
<td>( E4 )</td>
<td>None</td>
</tr>
<tr>
<td>EXP</td>
<td></td>
<td>( E5 )</td>
<td>None</td>
</tr>
<tr>
<td>GT</td>
<td></td>
<td>( E6 )</td>
<td>None</td>
</tr>
<tr>
<td>GE</td>
<td></td>
<td>( E7 )</td>
<td>None</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td>( E8 )</td>
<td>None</td>
</tr>
<tr>
<td>NE</td>
<td></td>
<td>( E9 )</td>
<td>None</td>
</tr>
<tr>
<td>LE</td>
<td></td>
<td>( EE )</td>
<td>None</td>
</tr>
<tr>
<td>LT</td>
<td></td>
<td>( ED )</td>
<td>None</td>
</tr>
<tr>
<td>NOT</td>
<td></td>
<td>( EF )</td>
<td>None</td>
</tr>
<tr>
<td>AND</td>
<td></td>
<td>( F\phi )</td>
<td>None</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>( F4 )</td>
<td>None</td>
</tr>
<tr>
<td>IMPL</td>
<td></td>
<td>( F5 )</td>
<td>None</td>
</tr>
<tr>
<td>EQUIV</td>
<td></td>
<td>( F6 )</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F7 )</td>
<td>None</td>
</tr>
</tbody>
</table>
14A. Parameter Notation

1 1'th User's label (p<1). Numbered in order of declaration. (2 bytes)

q q'th generated label (p<q). (2 bytes)

p p'th procedure label (p<p). Numbered in order of declaration. (1 byte)

k hierarchy number (p<k). (4 bits)

(k/l) k and l packed into two bytes with 4 bits for k and 12 bits for l.

(k/s) Stack address. k and s packed into two bytes with 4 bits for k and 12 bits for s, where s is the number of words along the notional stack from the start (s=0) of the stack for the current hierarchy k.

(k,L) k and L packed into two bytes with 4 bits for k and 12 bits for L, where L is the number of words of first order working storage (including parameter space and link data) for the hierarchy k and its constituent blocks.

(k/p) k and p packed into two bytes with 4 bits for k and 12 bits for p.

n n'th block level in current (or destination) hierarchy (1<n). (1 byte)

T T'th entry in Level Parameters (b) (0<T) (2 bytes)

Number of words along notional stack for some hierarchy k from s=0 to s= end of parameter space (1 byte).

N Total block nesting depth within current hierarchy. (1<N) (1 byte)

Constant 1 word for integer or boolean constants, containing the explicit constant. 2 words for real constants, containing the explicit constant.

'string' Basic symbol representation (unpacked - i.e. 1 byte per symbol) of the string, including the opening and closing string quote symbols.

- A one byte space

a Number of arrays in an array segment (1 byte)

b Number of dimensions of an array (1 byte)

r r'th failure for this module

S* Special location used in for statements. See Section 8A.

t Type of operation now replaced by the current LFAIL operation, according to the following code:-

R0 'X'20'! Result operation
A0 'X'40'! Address operation
S0 'X'60'! Statement operation
F0 'X'80'! Function operation
F0 'X'00'! Parameter operation

(1 byte)
Entryname  Up to 8 EBCDIC characters, left-hand justified and padded out with spaces (X'00') as necessary.

Identifier  Up to 8 Algol Basic Symbols packed into 6 bytes, each Basic Symbol occupying 6 bits. The significant 6 bit items are right-hand justified in the 6 byte field, padded out with zeros if necessary.
<table>
<thead>
<tr>
<th>1st Dig.</th>
<th>2nd Dig.</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FINISH</td>
<td>PE</td>
<td>RETURN</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>EIS</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TRR</td>
<td>TIR</td>
<td>TBR</td>
</tr>
<tr>
<td>3</td>
<td>TRC</td>
<td>TIC</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TRA</td>
<td>TLA</td>
<td>TMA</td>
</tr>
<tr>
<td>5</td>
<td>TRA</td>
<td>TIA</td>
<td>TBLA</td>
</tr>
<tr>
<td>6</td>
<td>SO</td>
<td>MRSF</td>
<td>HISF</td>
</tr>
<tr>
<td>7</td>
<td>INDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CF or FO</td>
<td>CTF</td>
<td>CFZ</td>
</tr>
<tr>
<td>9</td>
<td>CFF</td>
<td>CTFF</td>
<td>TRACE</td>
</tr>
<tr>
<td>A</td>
<td>TL</td>
<td></td>
<td>TLH</td>
</tr>
<tr>
<td>B</td>
<td>UJ</td>
<td>IFJ</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>PO</td>
<td>ABS</td>
<td>SIGH</td>
</tr>
<tr>
<td>D</td>
<td>PTR</td>
<td>PRF</td>
<td>PLF</td>
</tr>
<tr>
<td>E</td>
<td>+</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>NOT</td>
<td>NEG</td>
<td>AND</td>
</tr>
<tr>
<td>BASIC SYMBOL</td>
<td>ALGOL</td>
<td>ECMA</td>
<td>ALTERNATIVE</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>A-Z</td>
<td>A-Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a-z</td>
<td>A-Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\emptyset$-9</td>
<td>$'1\emptyset'$</td>
<td>^ or **</td>
<td></td>
</tr>
<tr>
<td>10 (subscript ten)</td>
<td>'REAL' etc</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>'***'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>'LT'</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>'NOT'</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>'('</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>(open string)</td>
<td>'('</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>+ (string space)</td>
<td>')'</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>')'</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>) (close string)</td>
<td>')'</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>'/'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>'='</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>'EQ'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>'OR'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>'GE'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>'IMPL'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>':='</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>(string tab)</td>
<td></td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>'GT'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>'EQUIV'</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>(string new line)</td>
<td></td>
<td>/</td>
<td></td>
</tr>
<tr>
<td># (string page change)</td>
<td></td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>$\text{segment}^*$</td>
<td></td>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

**N.B.** - is the 'break' or 'underline' character.