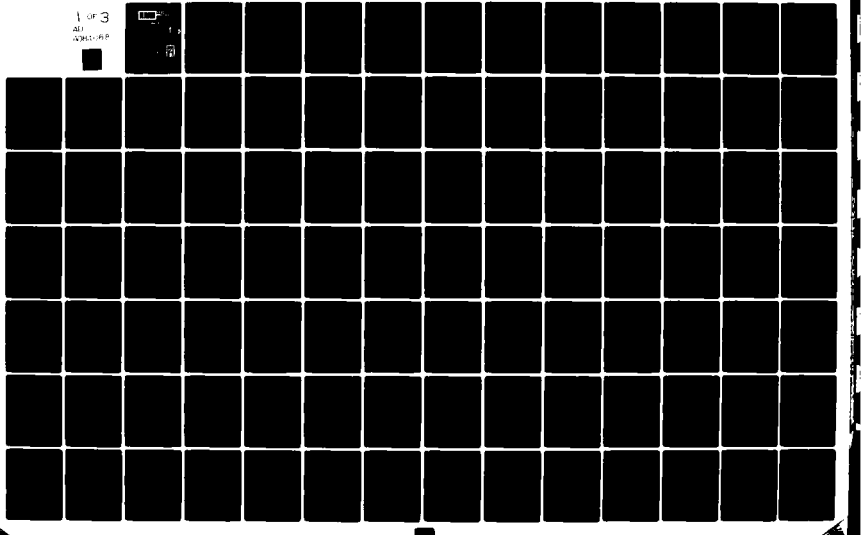
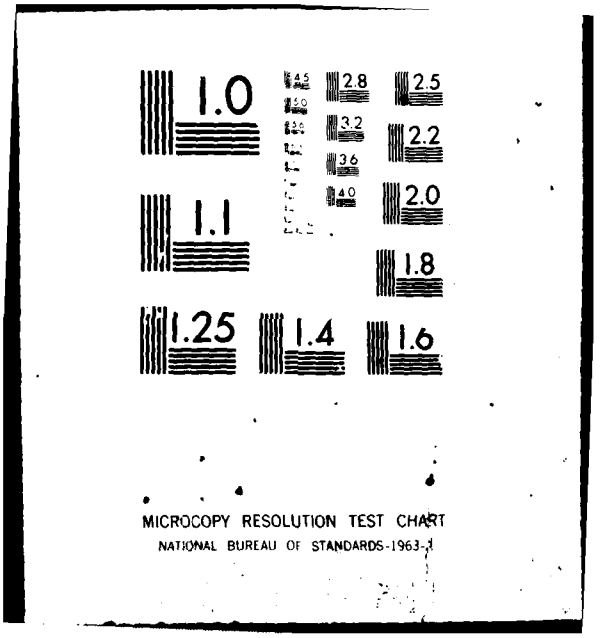
AD-A084 068 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
THE CORAL 66 COMPILER FOR FERRANTI ARGUS 500 COMPUTER.(U)
JUN 78 B GORMAN
UNCLASSIFIED RSRE-TN-799 DRIC-BR-67199 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RSRE LEVEL II

TECHNICAL NOTE

No. 799

THE CORAL 66 COMPILER FOR FERRANTI ARGUS 500 COMPUTER

Author: B. Gorman

June 1978

Royal Radar Establishment,
Procurement Executive,
Ministry of Defence,
Malvern, Worcs.

FOREWORD

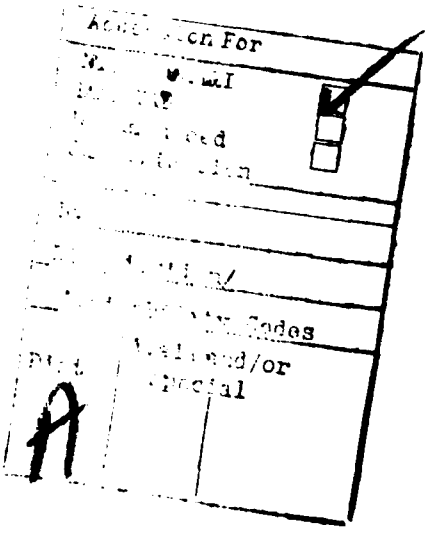
Looking back now, the compiler described in this document can be seen as an important stepping stone in the history of computer languages in Britain. It was commissioned by Ferranti Ltd for use in industrial control and automation projects where there had been a strong resistance to high level programming languages. Much was going to depend on the quality of the first compilers, and in particular the first Coral 66 compiler for the Argus. In the event this compiler proved to be outstandingly efficient and robust, which happily prevented attention being diverted from the main objective - increased productivity. I am assured by Ferranti that this objective was realized very quickly.

That it was possible to produce the Argus 500 compiler so expeditiously was due in no small measure to the software tools available to compiler writers at RSRE - tools which were quickly exploited and sharpened by Mr Gorman. RSRE, or RRE as it was, had long learned the wisdom of applying computer methods to the production of computer software.

Languages change and Coral 66 must eventually be modified (and ultimately displaced) as users become more demanding and compiler writers discover new methods of dealing with language problems. But the techniques which were used in making the Argus 500 Coral compiler are not obsolete, and should prove of particular interest and value to the many implementers of Coral 66.

P M WOODWARD

February 1977



CORAL 66 COMPILER FOR ARGUS 500

Part 1 - Description

Part 2 - Computer Printout

Introduction

This documentation is a working description of the Coral 66 compiler for the ARGUS 500 produced by RRE under contract from Ferranti Ltd.

It serves not only as a guide to the actual operation of the compiler, but also as an example of the use of RRE compiler building tools and techniques. These represent the culmination of many man years of research, as also do some of the important Algorithms used within the compiler. These techniques and Algorithms have been used in other compilers produced by RRE, and details have been, or will be, made freely available.

This description is not intended to be a specification of the Compiler, nor is it intended as a guide to compiler writing for general publication. It is hoped however that it contains sufficient information for the necessary understanding of the compiler operation required for its future maintenance.

Where any doubt is raised by, or ambiguity discovered in, the description of any section of the compiler, reference should be made to the actual program of the compiler, as this uniquely determines its operation. As the compiler itself is written in Coral, this should not prove to be too difficult.

The use of the compiler to compile itself makes it to a large extent self checking. Its very modular nature enables testing to be simplified as many modules are non interacting. The central routine, about which the whole compiler is built, uses a transformed syntax which has been exhaustively checked by syntax manipulation programs. These features should enable a high degree of confidence to be placed in the correct operation of the Compiler.

Part 1 of this document contains the detailed description of the individual modules forming the Compiler. This is preceded by notes on particular aspects of the techniques and standards used, and by an overall description of the compiler strategy, quoting the syntax rules in which it is embedded.

Part 2 consists of printouts of the actual program of the compiler and of the final syntax used. Each of these is accompanied by a list of all the identifiers used, giving the names of the pages on which they occur, the number of occurrences on each page, and the total number of occurrences.

Language Definition

This implementation is based on the Official Definition of Coral 66, (May 1970) and with the exception of two features, conforms to this definition. (Type D without Recursion)

The first deviation concerns bit numbering: Ferranti standard bit numbering is used. The second of these concerns Tables. On the ARGUS 500, it is inefficient to multiply the index by the number of words per entry each time a field is referred to. Therefore only the total size of a Table is specified, all indices requiring to be implicitly multiplied by the appropriate factor. As a consequence of this, individual fields may not be preset.

In addition to the Official Definition, the following extra features are incorporated:

- Shift Operators
- Exponentiation
- VALUE Procedures
- Repeated Procedure Calls
- Overflow Tests
- Implied Tests against Zero
- LABEL addresses as operands
- SPECIAL arrays
- Tracing Facilities
- Test Compilation
- Integers of specified significance
- Alternative use of square brackets
- Page Titles
- Alternative Literal and Octal constants
- Hexadecimal constants
- Identifiers commencing with £ and %
- Macro Identifiers terminating with !
- Optional Items preceded with ?

Addresses

Within the compiler, and also within the loader, addresses are handled as a 24 bit word in a consistent manner. Certain types of address will however only occur in a limited context. In general an address consists of two components: a base and an index. The base is in the range 0-7 and is held in 'BITS' [3,6] of the word, and the index in 'BITS' [14,10]. The eight different values of base refer to eight conceptual areas of core as follows:

Base	Area	Mnemonic
0	Absolute	A
1	Program	P
2	Data	D
3	Special data	S
4	Common	C
5	Library	L
6	External	E
7	Working constants	W

These areas are referred to by the first letter of the area name, and addresses within these areas by the area letter followed by one or more octal digits for the index. In the case of bases 1-4 the actual address is given by the sum of the relevant base plus the index. In the case of base 5 (Library) the index is the reference number of the library item. The index in the case of base 6 (External) is made up of two components, 'BITS'[5,10] being the number of the external relocation base, and 'BITS'[9,15] being 'OCTAL'(400) plus the displacement about that base. Base 7 (Working constants) is used to enable the loader to optimise the allocation of storage for fixed constants at load time. Within the compiler the index is always zero.

Bases 1, 2 and 3 always refer to locations within the current segment, the other bases referring to locations outside.

The remaining spare bits are used in the compiler and the loader to carry auxiliary information in specialised cases.

The most frequent of these within the compiler is where 'BITS'[10,0] are all set to 1's. In this case the value of the index is the number of an accumulator. An address of this type is never passed to the loader as it is automatically converted to an absolute address in the procedure OUTI. The value used to set these bits is given by AMARK.

Within the loader 'BITS'[3,3] may be used to hold a relativiser setting, these bits being in the correct position for setting the NWREL and NRREL registers.

DATA ALLOCATION

All non-overlayed data declared within a program segment is allocated locations within the D area of the Segment, and is addressed relative to the start of this area. The variable used within the compiler for this allocation is DATAMAX.

Two other variables are used, in conjunction with DATAMAX, for the allocation of anonymous locations used as temporary workspace. These variables are DATASTART and DATAPTR.

A flag PRESETOK is used to indicate whether presetting is allowed, preset items being output using the transfer address DTA.

The actions taken within the compiler concerned with data allocation are as follows:

A Before the compilation of a Program segment, the flag PRESETOK is set. The variables DATASTART, DATAMAX, and DTA are all set to the starting "address" of the data area. (ie Tag=D,index=0)

B At the start of a block, the current values of DATASTART, DATAMAX and PRESETOK are placed on the stack. DATAMAX is then set to the value of DATASTART.

C As the declaration of each variable, table, and array, is processed, it is allocated the address given by the current value of DATAMAX, which is then incremented by the number of words required by the item being declared.

D After each declaration, DATASTART is set to the value of DATAMAX.

E Before the first statement of a block, the flag PRESETOK is cleared.

F Before each statement within the block, DATAPTR is set to the value of DATASTART.

G Temporary locations, used within a statement, are allocated using DATAPTR, which is incremented by one each time. If its value exceeds that of DATAMAX, this is also incremented. Temporary locations allocated in this way may be re-used if it is known that they are vacant.

H At the end of a block, the values of DATASTART and PRESETOK are restored to their values held on the stack. DATAMAX is only reset if its stacked value exceeds its current value. As a result of this action, at the end of a Program segment, the value of DATAMAX will give the total data requirement of the segment. Similarly, after the declaration of a procedure, which is treated as a block even if it contains no declarations, the value of DATAMAX will have been incremented by the total data requirement of the procedure.

I Before a list of preset constants, the flag PRESETOK should be inspected. If it is clear, the presetting is illegal, and the list of constants must be ignored. If DTA is not equal to DATASTART, a directive is issued to the loader to skip the data transfer address forwards, and DTA is set to the value of DATASTART.

J After each preset constant, the number is rescaled to the scale required by the declaration. The number is only output if PRESETOK is set and DTA is less than DATAMAX. The value of DTA is incremented by one each time a constant is output.

Chaining and Jump Optimisation

In general, the destination of conditional and unconditional jump instructions generated by the compiler is not known at the time of their generation.

There are several solutions to this problem. One is to generate pseudo labels where the jump is not explicitly to a label ('IF' . . 'THEN' . . 'ELSE' and use an assembler to insert the correct addresses. This usually requires more than one assembler pass. Another method is to fill in the addresses by a further compiler pass. A third method is to use a "branch ahead" table, which records the addresses of incomplete jump instructions. When the destination is discovered, directives are output to fill in the required addresses. Unless this table is arbitrarily large it will become full at some time.

The chaining method used by the compiler has none of the above disadvantages. Consider the case of a simple forward jump to a label not yet set. The first jump instruction is output with a zero address field, and its address stored in the record for the label. If a second jump instruction to the same label is to be output, it is output with its address set to the address of the first jump, and the address of the first jump in the label record replaced with that of the second. This may be repeated an indefinite number of times without using any further storage space within the compiler or loader. This builds up a chain of jump instructions in the compiled program. When the destination is finally discovered, the compiler outputs a directive to set the addresses of the instructions on the chain to the required destination.

Further complexities arise in the single pass compilation of block structured languages. Consider the following example:

```
1       'BEGIN' 'FIXED' . . .
2           L: . . . . . .
3             'BEGIN' 'FIXED' . . .
4                   'GOTO' L; . . .
5                   'BEGIN' 'FIXED' . . .
6                         'GOTO' L; . . .
7                         'GOTO' L; . . .
8                   'END' LEVEL 3;
9                 L: . . .
10            'END' LEVEL 2;
11            'GOTO' L; . . .
12      'END' LEVEL 1;
```

When compiling the jump of line 4, it is not safe to assume that the jump is to the label L of line 2. In fact, although the information is not available at this point, it is a jump to the label L of line 9. Similarly, when compiling the jump of line 6, it is not safe to assume that the jump is to the label L of line 2, or even that it is the same label L referred to in line 4.

However, when compiling the jump of line 7, it is safe to assume that the reference to the label L is a reference to the same label used in line 6, because these both occur at the same block level. When the end of block level 3 is reached on line 8, it may be deduced that the label L referred to on line 4 is the same label L referred to on lines 6 & 7 because it has not been set at block level 3.

The setting of the label L on line 9 supplies the destination for the jumps of lines 4, 6, and 7. At the end of block level 2, line 10, this label goes out of scope, and any record of it may be safely discarded, because it has been set. The reference to the label L in the jump on line 11 may safely be taken to be a reference to the label of line 2 because these both occur at the same block level.

Thus, when searching for a label record, only the list pertaining to the current block level may be used.

These complexities introduced by block levels do not present any great problems when chaining is used. All cases may be handled with the following directives:

1 Set chain starting at N to current program address

2 Join chain M on to end of chain N

3 Set chain N to address A

The first directive is sufficient for non-block structured cases, and is used to set forward references within a block level. It may be seen that this directive is in fact a special case of the third.

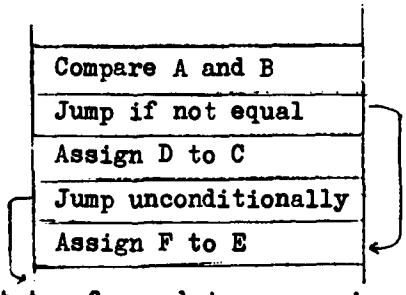
The second directive would be used in such cases as given at line 8 of the above example. Here a chain exists for block level 3 and also for level 2 (in its simplest form). As it is desirable to discard the level 3 record, the level 3 chain is joined on to the end of the level 2 chain, using a type 2 directive, only one reference within the compiler being retained. When the label is subsequently set, in line 9, a type 1 directive is used to set this composite chain.

If however the label setting of line 9 had been omitted, when the end of block level 2 was reached in line 10, it would be discovered that the label was already set at block level 1. In this case, the third type of directive would be used to set the chain to the address of the label in line 2.

Apart from jumps generated as the result of explicit 'GOTO' statements, there will be a considerable number of jumps generated "anonymously", not to specific labels, but implied by certain language features. The most obvious are those concerned with conditional statements. A statement such as:

'IF' A=B 'THEN' C←D 'ELSE' E←F;

might generate:



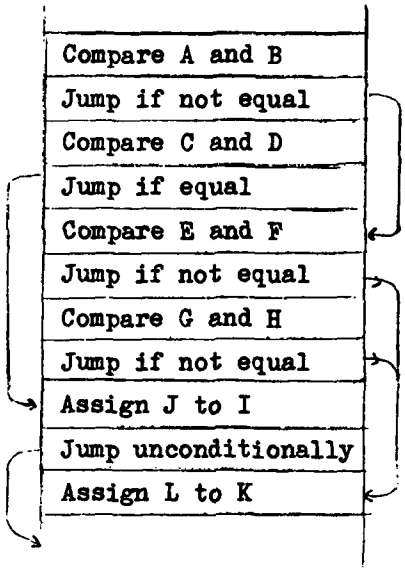
Here it may be seen that two forward jumps are involved, the first (conditional) jump skips over the consequence statement, and is due to be set at the commencement of the alternative statement. The second jump is used to skip over the alternative statement, and is due to be set at the commencement of the following statement.

Where a condition involves more than one comparison, the Official Definition states that the condition is to be evaluated from left to right only as far as is necessary to determine the overall truth or falsity. This implies that the condition is not to be evaluated as a Boolean expression, with only the final result being tested; but that each comparison must be tested individually, immediately after its evaluation.

Thus a statement such as:

'IF' A=B 'AND' C=D 'OR' E=F 'AND' G=H 'THEN' I←J 'ELSE' K←L;

might generate:

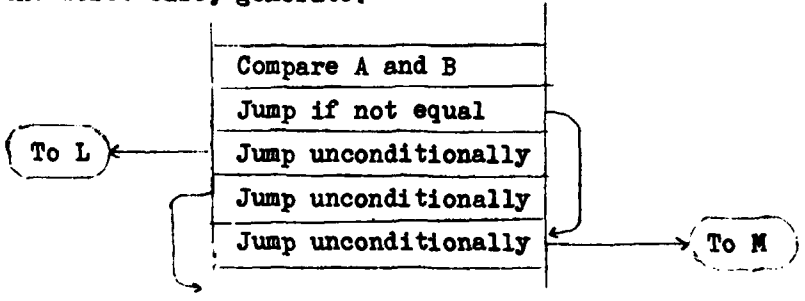


This example illustrates that, irrespective of the complexity of a conditional statement, only two chains are required. (per conditional depth). The first of these is the "skip" or "false" chain, which is used to skip over the consequence statement. The second being the "true" or "due" chain, which is used to jump directly to the consequence statement, and is due to be set at its commencement. The application of chaining to such cases should be obvious.

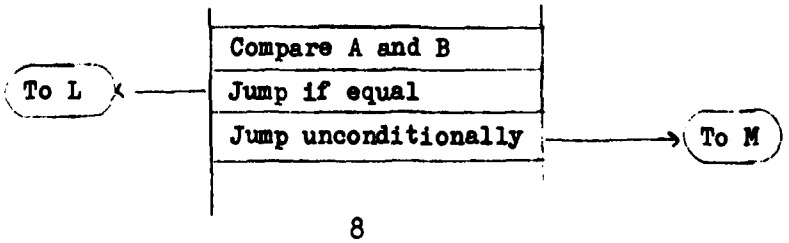
What is not so obvious however is how cases involving explicit conditional jumps may be compiled efficiently. As an example, a statement such as:

'IF' A←B 'THEN' 'GOTO' L 'ELSE' 'GOTO' M;

might, in the worst case, generate:



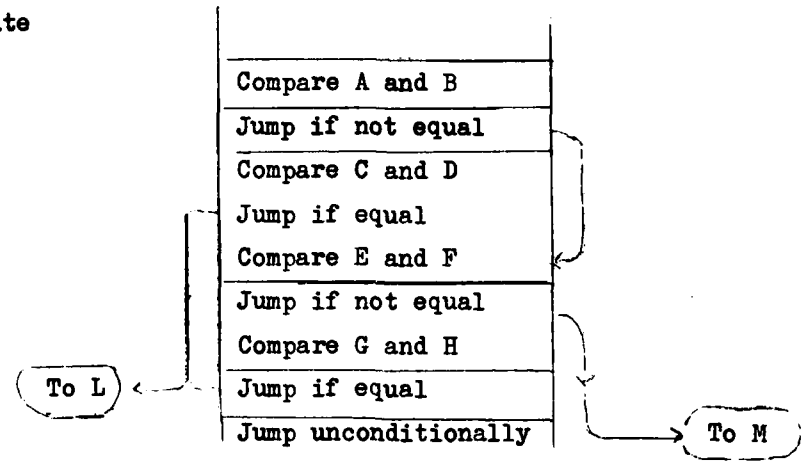
whereas it would be preferable to generate:



and in a complicated case, such as:

'IF' A=B 'AND' C=D 'OR' E=F 'AND' G=H 'THEN' 'GOTO' L 'ELSE' 'GOTO' M;

to generate



In the above example, the chains of jumps, which are set up before it is discovered that a jump to a label is required, may be set to, joined on to, or remembered as, the reference to the label at the appropriate block level.

This does not solve the problem of 'THEN' 'GOTO' optimisation. It may be inferred that it would be advantageous not to output the final test instruction on the occurrence of the symbol 'THEN', but to defer its generation. In the case of 'THEN' 'GOTO' LABEL its sense is required to be reversed, but not in the case of 'THEN' 'GOTO' SWITCH. Thus the syntactic detection of this case, although possible, becomes rather complicated.

Explicit conditions also occur following the symbol 'WHILE' in 'FOR' statements. It is interesting to note the similarity between the execution of the controlled statement if the condition is true, and the execution of the consequence statement in a conditional statement. Implicit jumps are also generated as a part of 'STEP' - 'UNTIL' elements, and also from the end of the controlled statement to the instructions to calculate and test a further value of the control variable.

Another case of implicit jumps is where procedure declarations occur at the head of a block, and require to be jumped around. Where however, this block is the outer block of a procedure, this jump may be omitted, as the entry point of the procedure has not yet been reached. In the Argus implementation, the location reserved to hold the pointer to the first executable instruction of a procedure needs to be set when this point is reached.

In the case of typed procedures, it is desirable to check that the procedure has at least one 'ANSWER' statement, and that a return to the point of call is only made by means of such a statement. The return to the point of call by obeying the implicit return instruction at the end of the procedure body is only applicable in the case of untyped procedures. In the Argus implementation, a single return instruction may be generated after each 'ANSWER' statement, except where 'PROCEDURE' 'TRACE' is required; in which case all 'ANSWER' statements except one occurring as the last statement of the body require to be followed by a jump to the tracing instructions generated at the end of the procedure.

The scheme used by this compiler requires two chains at each conditional level, referred to by the variables SKIPCHAIN and DUECHAIN within the compiler. It also makes use of a variable, STATUS, which indicates the context in which a statement occurs.

At the start of each statement, DUECHAIN holds a chain of jumps, which may in fact be null, due to be set before the start of the compilation of the statement; SKIPCHAIN holding a chain of jumps bypassing the statement. At this point, the values assigned to STATUS indicate the contexts set out below.

STATUS=0

Control is passed normally from the preceding statement. If DUECHAIN is non zero, the statement is also entered by means of a jump. (The contents of the accumulators being undefined)

STATUS=1

Control is not passed normally from the preceding statement. This statement can only be entered by a jump. If DUECHAIN is zero, the statement cannot be entered.

STATUS=2

The preceding statement was an 'ANSWER' statment, and requires either an exit instruction or jump to the end of the procedure (using EXITCH) depending on whether trace is required. If DUECHAIN is zero, the statement cannot be entered.

STATUS=3

The preceding symbol was 'ELSE', and an unconditional jump around this statement is required before DUECHAIN (which will be non zero) can be set.

STATUS=4

The preceding symbol was 'THEN', (or 'THEN' 'ELSE' together) and a conditional jump around this statement is required before DUECHAIN (which may be zero) can be set.

At the beginning of the compilation of a block which does not form the body of a procedure, both STATUS and DUECHAIN will be zero, it being assumed that such blocks are actually entered at their head.

If a procedure declaration occurs whilst STATUS is zero, an unconditional jump, with an initially zero address part, is output, its location being recorded in DUECHAIN. This is the instruction to skip over procedure declarations. Its use could be avoided, but only at the cost of completely re-ordering a segment, which is an extremely complicated operation with a single pass compiler.

At the start of the procedure declaration, after the skip over instruction has been output if required, the current value of DUECHAIN is stacked. STATUS is then set to 1, and DUECHAIN set to the location of the pointer to the procedure. (In the case of Library procedures, to the Library "address", with a sign marker for the convenience of the loader.) After the declarations, if any, at the head of the procedure, DUECHAIN is set, STATUS is set to zero, and the instructions to store the link and parameters generated as required.

At the end of the procedure declaration, DUECHAIN is restored to its stacked value, and STATUS is set to 1. When the actual start of the block is found, DUECHAIN will be set, thus completing the skip over instruction.

At the start of each statement, the value of STATUS is required to be inspected, and the chain DUECHAIN is required to be set. Except in the case of statements of the form 'GOTO' Label, this is carried out by the procedure STATUS TEST, which is called by the compiling actions STATUSCHECK and STATUSCODE. This procedure outputs the required jump instruction if the value of STATUS is greater than one, and resets STATUS to zero. If DUECHAIN is non-zero, STATUSTEST sets the chain to the current program address, and resets DUECHAIN to zero.

Most statements will leave STATUS set to zero. Answer statements will however set it to 2, and most cases of 'FOR' and 'GOTO' statements will set it to 1.

A conditional statement will initially set the value of STATUS to 4. The current value of SKIPCHAIN is stacked, and SKIPCHAIN is set to zero in readiness for the following condition. (DUECHAIN will at this point be zero.) After each comparison (or overflow test), FUNCTION and ACCUMULATOR are set to the appropriate values for a conditional jump instruction, the jump occurring if the "result" of the comparison is false.

If the comparison is followed by the symbol 'AND', this jump is output with its address on the SKIPCHAIN. If however the comparison is followed by the symbol 'OR', the type of the test is reversed, the instruction output with its address on the DUECHAIN, the SKIPCHAIN set to the following comparison, and SKIPCHAIN cleared.

Following the symbol 'THEN', the consequence statement will be processed. Provided that this statement is neither null, nor of the form 'GOTO' Label, this will present STATUSTEST with a value of STATUS of 4. In this case the conditional jump instruction will be output with its address on the SKIPCHAIN before DUECHAIN is inspected.

If however, the consequence statement is of the form 'GOTO' Label, the type of the test instruction is reversed. This is then output, using the chain for the label as its address, DUECHAIN being linked onto the end of this chain (or set, as appropriate). This case leaves DUECHAIN clear, and STATUS set to 5.

If the symbol 'ELSE' follows, preparation is made for the alternative statement. The values of SKIPCHAIN and DUECHAIN are exchanged, thus the skip over the consequence statement being due to be set at the start of the alternative statement. If STATUS is zero, it is set to 3 to indicate that a jump to skip over the alternative statement will be required. If however the value of STATUS is 5, this jump is not required, and STATUS is set to zero. If the value of STATUS is 4, the consequence statement was null, the type of the test instruction is reversed, and STATUS left set to 4.

The alternative statement will usually present STATUSTEST with a value of STATUS of 3. In this case, an unconditional jump instruction, with its address on the SKIPCHAIN, will be output. If however, the alternative statement is of the form 'GOTO' Label and STATUS is 3, no instructions will be output. In this case the DUECHAIN is set to the address of, or added onto the chain for, the label; STATUS being reset to zero.

At the end of the conditional statement, the value of STATUS will only be 4 if the consequence and alternative statements are both void. The test instruction is output only if it is an overflow test. If the value of STATUS is greater than 2, STATUS is set to zero. The SKIPCHAIN is then joined on to the end of the DUECHAIN, and SKIPCHAIN reset to its stacked value.

'FOR' statements also stack the value of SKIPCHAIN, this chain being used in conditions and tests within the for-list. Except in the case where the for-list is a single step-until element consisting only of three constants, the SKIPCHAIN is used to exit from each for element in turn, as it becomes exhausted, and finally from the for statement.

At the end of the for statement, the value of SKIPCHAIN is assigned to DUECHAIN, and SKIPCHAIN reset to its stacked value. Where the simple step-until case has been detected, STATUS will be left set to zero. Otherwise it will be set to 1, except in certain cases where it will be left set to 2.

('FOR' . . (,) . . 'WHILE' . . 'DO' . . 'ANSWER' . .)

The cases of a statement of the form 'GOTO' Label occurring as consequence and alternative statements has been mentioned above. Where a statement of this form occurs, and STATUS is 1 or 2, no instruction is output, the DUECHAIN being set to the address of, or joined on to the chain of, the label. Where STATUS is zero, an unconditional jump to the label is output, and STATUS is set to 1.

The compilation of a 'GOTO' Switch statement is always preceded by the execution of STATUSCHECK. After the required instructions have been output, STATUS is set to 1.

At the end of the body of an untyped procedure, a return instruction is output unless STATUS is 1 and DUECHAIN is zero. In this case the procedure has been left by means of a goto statement, and a return to the point of call is not required.

A typed procedure must have at least one 'ANSWER' statement. This will leave STATUS set to 2 and/or EXITCH set non-zero. If, at the end of the procedure, DUECHAIN is non-zero or STATUS is zero, an attempt is being made to return to the point of call other than by an 'ANSWER' statement.

At the end of a Program segment, a final stop instruction is required, unless DUECHAIN is zero and STATUS is 1. In this case the program is effectively terminated by a jump.

Conditional expressions require the values of both SKIPCHAIN and DUECHAIN to be stacked. STATUS optimisation is not applicable in this case, so the value of STATUS is left unchanged.

The Compiler-Loader Interface

The output of the compiler is a form of relocatable binary. In the basic form of the compiler, this is punched on paper tape. The unit of information output by the compiler ('PROCEDURE' OUT5) is a five character group, each character requiring six bits. These characters are output as if they were legible characters, and so the standard internal-external code conversion routines may be used. The first character of the group is a directive or tag (T) character, the remaining four characters being treated as a 24 bit word, (W).

Where the tag indicates that the word (W) is to be stored as part of the current segment, the three most significant bits of the tag (T) specify with which transfer address the word is to be loaded, and the three least significant bits specify with which base the word is to be relocated. It should be noted that Program instructions are initially stored with the address in the least significant fourteen bits, and subsequently given ten places of left cyclic shift.

Most of the tag values used for directives require more than one word to specify the action required. Thus the requirement for a multi-length group arises. This is satisfied by allocating tag 0 as a (prefix) continuation tag. When a continuation tag is read the current word (W) is stored in an array, and the index used in referring to this array incremented, thus variable length directives may be used. This is required in the cases where a string is passed as part of a directive. The first location of this array is referred to as C, and the subsequent locations as C_1, C_2 etc.

Where an address is specified as part of a directive, the form of the address is exactly the same as that used within the compiler. Before use, the address is relocated with the base specified by bits 6-8 of the word.

List of Tags

0 Continuation, store W in C_n , n←n+1
1 Size Block, W=checksum, C = Program, C_1= Data, C_2 = Special
2 Display message, W=DISP, C=string
3 Start of common check, W=common checksum
4 Library Variable Block, W=checksum, C=Lib no, C_1=Value, C_2=Identifier
5 Library Check, W=LCHQ
6 Entry Block, W=NTER
7 Stop Block, W=STOP
8 Store Program, relocate as Absolute
9 Store Program, relocate as Program
10 Store Program, relocate as Data
11 Store Program, relocate as Special data
12 Store Program, relocate as Common
13 Store Program, relocate as Library
14 Store Program, relocate as External
15 Store Program, relocate as Working constant, C=constant
16 Store Data, relocate as absolute
17 Set chain W to current program address unless W -ve, in which case set entry point for Library procedure W.
18 Set chain W to address C
19 Store Data, relocate as Special data
20 Join chain W on to end of chain C
21 Set common W to procedure C
22 Set common W to switch C
23 Set common W to label C
24 Store Special data, relocate as Absolute
25 Store Special data, relocate as Program

```
26    Store Special data, relocate as Data
27    Store Special data, relocate as Special data
28    Store Special data, relocate as Common
29    Store Special data, relocate as Library
30    Store Special data, relocate as External
31    Skip Data transfer address forward to W
32    spare
33    Print address W and Identifier string C
34    Print instruction W and comment string C
35    Change instruction W to 4 JCS
36-38 spare
39    End of segment, W=segment checksum. (C=Common checksum)
40-55 spare
56    Check Common location W, C=Identifier string
57-60 spare
61    Program Segment, W=Common checksum, C=Identifier string
62    Library Segment, W=Library number, C=Identifier string
63    Common Segment, W=0, C=(COMMON)
```

List of Compiler Sections Generating Tags

```
 0      OUTCONT
 1      FINISHSEG
 3      COMOFF
 4      SETLIBVAR
 7      TTAIL
 8-16   OUT24
17      SETCHAINTOPTA
18      JOINCHAINS & ENDFOR
19      OUT24
20      JOINCHAINS
21-23   ENDPROG
24-30   OUT24
31      SKIPDTA
33      DIAG
34      OUTI
35      SETRT
39      ENDSEG
56      COMOFF
61-63   STARTSEG
```

Size Blocks (Tag 1)

The segment size block, which although only output after the segment has been compiled, must be read by the loader before the header of the segment. The format of a size block is as follows:

W	Checksum for block
C	Program Size
C_1	Data Size
C_2	Special Data Size

A size block must be followed by a segment header block (tags 61-63).

Program Header Block (Tag 61)

W Common checksum

C Segment Identifier String

Only if a segment has been compiled as a single segment program, without a Common communicator having been read, will the word W be zero. If the word W is non zero, it must be identical to the Common checksum of the previously loaded Common segment.

Library Procedure Header Block (Tag=62)

W Library reference number of Procedure

C Procedure Identifier String

If there is an outstanding requirement for the procedure, the segment should be loaded. Otherwise it should be ignored, by scanning forward until an end of segment tag (39) is read.

Common Segment Header Block (Tag=63)

W Zero

C "(COMMON)"

The Common checksum must be zero when the Common segment is loaded, ie only one Common segment is allowed to be accessed by a Program segment. The Common checksum is set by the C of the end of segment directive, and subsequently cleared if a successful Common check has been made.

End of Segment Directive (Tag=39)

W Checksum for RLB tape

C Only present at end of Common segment, and gives Common checksum

This directive indicates the end of a segment. In the case of a Common segment, the Common checksum is in C, and subsequent segments referring to Common must have the same checksum in their header. In the cases of Program and Library segments the Program area is to be given ten places of left cyclic shift.

Library Variable Block (Tag=4)

W Checksum for block

C Library reference number of Variable

C_1 Preset value of variable

C_2 Identifier String

If there is an outstanding requirement for the variable, this should be set to the initial preset value.

Common Check Header Block (Tag=3)

W Common Checksum

This block is the header of a Common Check tape. The Common checksum in W must match that set up by the previously loaded Common segment. This header is followed by directives (Tag=56) which check that the references within the Common area to Procedures, Switches, and Labels have been supplied. The format of these directives is as follows:

W Location to be checked

C Identifier string

Provided that all the checks have been successful, when the end of segment directive (Tag=39) is read, and its checksum for the tape found to be correct, the Common checksum is cleared. Unless this is cleared, the program cannot be entered. (or another Common loaded)

Instructions referring to Constants (Tag=15)

W Instruction

C Constant

It is left to the loader to optimise the following:

1 The use of the functions 04-07

2 Allocation of storage for constants

In the first case, if the constant is "short", after negation if negative, and the function is in the range 00-03, the function is changed, and the constant inserted into the address field.

In the second case, only one copy of each constant will be stored, irrespective of the number of references to it in different segments (including Library). The address of the constant is inserted into the instruction, and the function part unaltered.

Change Instruction Directive (Tag=35)

W Address of instruction

This directive is used in certain cases of 'FOR' statement. The instruction at location W is to be changed from an 0 JZE to a 4 JCS and the address set to the current Special Data transfer address. (See SETRT)

Print Address Directive (Tag=33)

W Address

C Identifier String

These directives are generated by DIAG if the 'LEVEL' is 2 or 3. These enable "milestones" to be printed at load time with their actual address.

Print Instruction Directive (Tag=34)

W Address of Instruction

C Comment string

These directives are generated by OUTI if the 'LEVEL' is 3. These enable instructions to be printed out in absolute form together with a commentary. As the address of the instruction may not be set until the end of the segment, this requires a second pass of the segment block.

References to Library (Tags=13,29)

W Least significant 14 bits contain reference number.

The Library reference number should be replaced by the address of the (pointer to) the Library item.

References to External Addresses (Tags=14,30)

W Least significant 14 bits contain external "address"

The address requires the addition of the specified external base to the displacement.

Paper Tape

The paper tapes punched by the compiler are made up from combinations of the items given below. All tapes are in ISO code with even parity, and except for blank and erase characters, contain only the 64 "printing" characters, corresponding to the internal code representations 0-63.

A Stop Block and Erases

The five characters 7STOP followed by 20 erases.

B Blank

Ten inches of blank tape.

C Size Block

Gives space requirements for following segment. Comprised of 20 characters, and includes checksum.

D Segment Block

Gives the relocatable binary form of a Program, Common, or Library segment; and includes checksum.

E Common Check Block

Enables a check to be made to ensure that all references within a common segment have been supplied.

F Library Data Block

Gives the Identifier, reference number, and preset value of a Library variable; and includes checksum.

In addition, the following items may be used.

G Library Check Block

The five characters 5LCHQ.

H Entry Block

The five characters 6NTER

Program Segments

The compiler punches the following items:

B D B A B C B A B

The size block (C) should be spliced on to the head of the tape, and the second stop block discarded; thus re-ordering the items as follows:

B C B D B A B

The lengths of blank tape (B) between other items are ignored, and their length is unimportant. Where several segments are to be loaded sequentially, they may be spliced together, and all stop blocks (A) except the final one, discarded.

Common Segments

The compiler punches the following items:

B D B A B C B A B E B A B

As with Program segments, the size block (C) should be spliced on to the head of the tape, thus giving:

B C B D B A B

This tape should be loaded before any Program segments referring to it are loaded. When all these Program segments have been loaded, the remainder of the tape output for the common segment:

B E B A B

should be loaded. This is the common-check tape, and unless this check is satisfactorily completed, the loader will not allow the program to be entered.

Library Segments

The compiler punches the following items:

B D B A B C B A B

The size block (C) should be spliced on to the front of the segment block (D), both stop blocks being discarded. The resulting tape:

B C B D B

should be spliced on to the front of the library tape.

Library Variables

The compiler punches the following items:

B F B A B

The stop block should be discarded, and the items:

B F B

should be spliced on to the front of the library tape.

Library Tape

The library tape should be built up incrementally, each Procedure referring only to previously compiled Procedures and Variables. These must occur on the relocatable binary library tape in the reverse order to that of their compilation. This is why new segments must be placed on the head of the tape. It is convenient for a library tape to end with the items:

B G B H B

This checks that the library is complete, and causes the program to be entered at the start of the (first) Program segment. (Where a complete Library is to be compiled, a utility program is available for processing the compiler output without the need for tedious splicing.)

Compiler Data Structures

The workspace required by the compiler consists of:

1 A large table, STACK

2 Global variables and arrays

3 Groups of variables specific to certain language features.

4 Local variables for procedures and labelled blocks.

The table STACK, is declared as having only 39 words of store. This is the fixed part which is directly referenced, usually by means of overlayed identifiers. The variable part, which should exceed 3K words if the compiler is to be compiled, is always referred to by pointers. These pointers usually pointing to chains of records. The actual area occupied by the variable part of the stack is set at load time, the three "area" pointers MCSTART, STACKSTART and STACKTOP being set up by the use of Externals. These three pointers refer to areas of core relative to the start of the stack.

The area from MCSTART to STACKSTART is used for macro expansion, its size depending upon the anticipated use of this facility. The area from STACKTOP downwards is used to hold records of labels, this area being shared with the main stack which starts at STACKSTART, and increases upwards. A check is made to ensure that these two stacks do not overlap.

With the exception of Arithmetic Operand records, all records held on the main stack are chained. The first word of each record holding a pointer, relative to the base of the stack, which refers to the succeeding record of the same type. At any one time there may be many chains of records on the stack, and extreme care is required in removing them in the reverse order to that in which they were placed on the stack. Fortunately, because of the nature of the language being compiled, this does not present much difficulty.

Chained records are always placed on the stack using the procedure ONSTACK. This has two parameters, the number of words of the record, and the location in core of the first word. This procedure uses GRABSTACK to allocate the required number of words, which uses STACKCHECK to ensure that this is available. When the record has been placed on the stack, by MOVE, the first location in core from which the record has been moved, is updated to point at the record. Records may be moved off the stack, back to fixed locations in core by the use of OFFSTACK. Note however that this procedure assumes that any space anove the record on the stack is no longer required. There is automatically no stack compaction, nor is it needed to languages of the complexity of Coral.

Groups of variables associated with certain (recursive) language features are moved to and from the stack as the depth of (recursion of) the use of the feature increases and decreases. These groups commence with the identifiers:

BLOCKCHAIN,LABCHAIN,PROCCHAIN,IFCHAIN,EXPRCHAIN & FORCHAIN

The features with which these groups are associated may be deduced from their identifiers.

Not all groups on a given chain are always of the same length. In the case of EXPRCHAIN for example: when the expression level is raised by one, at an opening round bracket, five words are placed on the stack, on the expression chain. When a procedure call is detected however, eight words are placed on the stack on the same chain. The extra expression level being required at the procedure call level for the evaluation of parameters.

A very important chain is the DECLIST chain of Identifier Specification records. This holds the specifications of all identifiers (except those of labels local to a segment) which are in scope. These records consist of five words plus the Identifier string. During the processing of declarations these records are built up incrementally in the fixed part of the stack. These are moved onto the main stack, usually in their completed form, by the compiling action NEWNAME. The details of this type of record are given in the declaration of STACK, this being the only type of record, (apart from Arithmetic Operand records, which share the same structure) which is accessed whilst being held on the stack.

Advantage is taken of the effect of OFFSTACK of reclaiming all space occupied above the record to which it is applied, in the actions at the end of a block. This automatically reclaims space occupied by Identifier Specification records of identifiers which go out of scope at that point. Because of the complexities introduced by labels, the identifier records of labels are kept in a separate stack, compaction being required at the end of a block. As no type information is required to be kept, these records are shorter.

Where an Identifier Specification record is that of a procedure, the PARAMSPEC word points at a Parameter Specification record, which usually follows the procedure's Identifier Specification record. This record is of between one and seven words in length. It holds a summary of the parameter requirements of a procedure, in the form of the TYPEBITS word of each parameter, and is terminated by a negative word. Thus if a procedure requires no parameters, its Parameter Specification record consists of a single negative word. As a procedure cannot have more than six parameters, this record cannot exceed seven words in length.

Arithmetic Operand records are created for each primary during the compilation of an expression. These records are five words long, the last four words having the same structure as Identifier Specification records. The first word of these records is not a chain pointer (CHAIN), but a pointer to a string (SPIEL), which is used as the comment in a Level 3 listing.

Where a primary is an identifier, possibly subscripted or with parameters, the Arithmetic Operand record is created by copying the relevant four words from the Identifier Specification record, and setting the first word to point to the identifier string in the Identifier Specification record. (By LOOKUP)

Where a primary is not derived from an identifier, for example a constant, the Arithmetic Operand record is synthesised from the available information, and the string pointer set to point to an appropriate fixed string (eg "(CONST)").

Arithmetic Opernad records are placed on the top of the stack. Only the two records currently at the top of the stack are referred to during the generation of instructions for arithmetic operations. These two records are referred to by means of the pointers LH and RH, which point to the left hand and right hand operands of the current operator. When an Arithmetic Operand is added to, or deleted from, the stack, the pointers LH and RH are updated.

Whilst generating the instructions for an arithmetic operation, (eg *) the information held in the records referred to by LH and RH is utilised. At the completion of this, the top (RH) record is deleted, and the remaining (LH) record modified to become the record of the result of the operation. Thus, after the compilation of an arbitrarily complicated expression, only one Arithmetic Operand record remains, this giving the details of the result.

Procedure calls cause one record to be set up on the occurrence of the procedure identifier, and also one record for each parameter. When the call is completed, the records for the parameters are deleted, and the record for the procedure modified to give the details of the result.

Subscripts are processed term by term. The compiler actions operating on the record of the current term, and the record of the variable being subscripted. After each term has been processed, its record is deleted. This optimisation is also used for shift operations, in which case a temporary record, similar to that of an anonymous reference, is used for the operand giving the number of places of shift.

LIBRARY ENTRANTS REQUIRED BY CORAL COMPILER

A For Trace and Diagnostics

'LIBRARY' 'PROCEDURE' %ENDOFPROG/1;

This is called at the end of every program segment which does not finish with a GOTO statement.

'LIBRARY' 'PROCEDURE' %UNSETLABEL/2('VALUE' 'INTEGER' LABEL);

This is called when a jump is attempted to a label which has not been set. The name of the label is passed in the form of a string as the parameter.

'LIBRARY' 'PROCEDURE' %LABELTRACE/3('VALUE' 'INTEGER' LABEL);

This is called when a label in the scope of a 'LABEL' 'TRACE' directive is jumped to or passed through. The name of the label is passed in the form of a string as the parameter.

'LIBRARY' 'PROCEDURE' %ENTERPROC/4('VALUE' 'INTEGER' PROC);

This is called when a procedure in the scope of a 'PROCEDURE' 'TRACE' directive is entered. The name of the procedure is passed in the form of a string as the parameter.

'LIBRARY' 'INTEGER' 'PROCEDURE' %EXITPROC/5('VALUE' 'INTEGER' PROC);

This is called when a procedure in the scope of a 'PROCEDURE' 'TRACE' directive returns to the point of call. The name of the procedure is passed in the form of a string as the parameter. This procedure must 'ANSWER' %ANSDUMP to restore the result in cases where the procedure being traced delivers a result.

'LIBRARY' 'PROCEDURE' %ANSWER/6('VALUE' ANSWER:SCALE);

This procedure is called when a procedure in the scope of a 'PROCEDURE' 'TRACE' directive delivers an answer. The first parameter is the result of the procedure and the second parameter its type-scale. The answer must be assigned to %ANSDUMP so that it may be restored by %EXITPROC.

'LIBRARY' 'INTEGER' 'PROCEDURE' %ASSIGN/7('VALUE' RESULT:SCALE; 'VALUE' 'INTEGER' VAR)

This procedure is called after the value of an assignment statement in the scope of an 'ASSIGNMENT' 'TRACE' directive has been computed, and before the actual assignment takes place. The first parameter is the value calculated and the second parameter its type-scale. The third parameter is the name of the variable to which the assignment is to be made, this is passed in the form of a string. Where the assignment is to be made anonymously the string is "(ANON)".

This procedure must 'ANSWER' RESULT; to restore the value.

'LIBRARY' 'INTEGER' %ANSDUMP/8

This variable is used to store the result of a procedure between calls of %ANSWER and %EXITPROC.

```
'LIBRARY' 'PROCEDURE' %FORTRACE/9('VALUE' VAL:SCALE; 'VALUE' 'INTEGER' VAR);
```

This procedure is called each time the controlled statement of a for-statement within the scope of a 'LOOP' 'TRACE' directive is entered. The first parameter is the value of the control variable, and the second its type-scale. The third parameter is the name of the variable in the form of a string.

B Floating Point Arithmetic

The following procedures preserve the setting of the overflow indicator on entry, and reset it to its previous setting on exit; unless the operands cause the procedure to deliver a result which lies outside the prescribed range, in which case the overflow indicator will be set on exit, and the result will be zero.

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPFLOAT/10('VALUE' NUMBER:SCALE);
```

This procedure is used to convert 'INTEGER' and 'FIXED' values to 'FLOATING', in standardised form. The first parameter is the number to be converted, and the second its type-scale.

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPMULT/11('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point multiply operation, delivering as its result the floating point value of the first parameter times the second parameter. (A*B)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPADD/12('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point add operation. (A+B)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPSUB/13('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point subtract operation. (A-B)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPDIV/14('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point divide operation. (A/B)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRAISE/15('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point exponentiate operation. (A↑B)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRAISEi/16
                  ('VALUE' 'FLOATING' A;'VALUE' 'INTEGER' I);
```

This procedure performs the floating point exponentiate operation to an integer power. (A↑I)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRSUB/17('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point subtract operation with the operands interchanged. (B-A)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRDIV/18('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point divide operation with the operands interchanged. (B/A)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRAISE/19('VALUE' 'FLOATING' A,B);
```

This procedure performs the floating point exponentiate operation with the operands interchanged. (B↑A)

```
'LIBRARY' 'FLOATING' 'PROCEDURE' %FPRRAISEi/20
                    ('VALUE' 'INTEGER' I; 'VALUE' 'FLOATING' A);
```

This procedure performs the floating point exponentiate operation to an integer power, the operands being in the reverse order to those of %FPRAISEi. (A↑I)

```
'LIBRARY' 'INTEGER' 'PROCEDURE' %FPFIX/21
                    ('VALUE' 'FLOATING' A; 'VALUE' INTEGER' SCALE);
```

This procedure converts the floating point number supplied as the first parameter to the scale specified by the second parameter.

C NON-STANDARD LIBRARY PROCEDURES USED BY FLOATING POINT ARITHMETIC

'LIBRARY' 'PROCEDURE' %FPNORM/22; @7-number,@6-exponent,@4-link+overflow marker.

This procedure normalises and packs floating point numbers. On exit the floating point number is in @7, and overflow is only set if the sign bit of @4 was 1 on entry. If however, the number is out of range, %FPERROR is called.

'LIBRARY' 'PROCEDURE' %FPERROR/23; @4=link

This procedure is entered by all floating point procedures where the operands or the results lie outside the specified range. This procedure sets overflow and clears @7; (This procedure could be revised to have a parameter specifying the type of error, this could be either a simple integer or a string).

'LIBRARY' 'PROCEDURE' %FPLOGCOM/50; @7=floating point number,@5=link

This procedure evaluates the logarithm to base e of the floating point number supplied in @7 as a'FIXED' (18,11) number in @7. If overflow is set on entry the sign bit of @4 is set to 1. This procedure does not use @6.

'LIBRARY' 'PROCEDURE' %FPEXPCOM/52; @7=number, @4=link+overflow marker,@3=-scale.

This procedure evaluates e to the number given in @7 after fixing the number, using the scale given in @3. This procedure exits to %FPNORM without altering @4.

D OTHER FLOATING POINT PROCEDURES

(See note on section B regarding overflow)

'LIBRARY' 'FLOATING' 'PROCEDURE' LOG/51('VALUE' 'FLOATING' A);

This procedure evaluates $Log_e(A)$, where A>0.

'LIBRARY' 'FLOATING' 'PROCEDURE' EXP/53('VALUE' 'FLOATING' A);

This procedure evaluates e^A.

'LIBRARY' 'FLOATING' 'PROCEDURE' EXPM/54('VALUE' 'FLOATING' A);

This procedure evaluates e^{-A}.

'LIBRARY' 'FLOATING' 'PROCEDURE' SIN/61('VALUE' 'FLOATING' A);

This procedure evaluates sine(A), where A is in radians.

'LIBRARY' 'FLOATING' 'PROCEDURE' COS/62('VALUE' 'FLOATING' A);

This procedure evaluates cosine(A), where A is in radians.

'LIBRARY' 'FLOATING' 'PROCEDURE' ARCTAN/63('VALUE' 'FLOATING' A,B);

This procedure evaluates $\tan^{-1}(A/B)$. The result is in radians in the range $\pm\pi$.

E OTHER PROCEDURES DELIVERING A NUMERIC RESULT

'LIBRARY' 'VALUE' 'PROCEDURE' SQRT/40:OUTSCALE('VALUE' X:INSCALE);

This procedure evaluates the square root of X, where X>0. The result is scaled to the scale required by OUTSCALE. Overflow will only be set on exit if it was set on entry, if X is negative, or the result cannot be rescaled to the required scale.

'LIBRARY' 'FIXED'(24,23)'PROCEDURE' FIXSIN/65('VALUE' 'FIXED'(24,23) X);

This procedure evaluates sine (πX). (ie X is scaled in half revs) Overflow is cleared on exit. As a result of +1.0 would cause overflow, sine$(\pi/2)$ is represented by $1-2^{-23}$. Similarly sine$(-\pi/2)$ is represented by $-1+2^{-23}$.

'LIBRARY' 'FIXED'(24,23)'PROCEDURE' FIXCOS/66('VALUE' 'FIXED'(24,23) X);

This procedure evaluates cosine(X) using sine((X+)).

'LIBRARY' 'FIXED'(24,23)'PROCEDURE' RANDOM/71;

This procedure delivers random numbers in the range 0 to 1.

'LIBRARY' 'INTEGER' %RANDOMINT/70← 262143;

This is used as workspace by RANDOM. Its initial setting determines the sequence of random numbers produced by RANDOM.

READER

Introduction

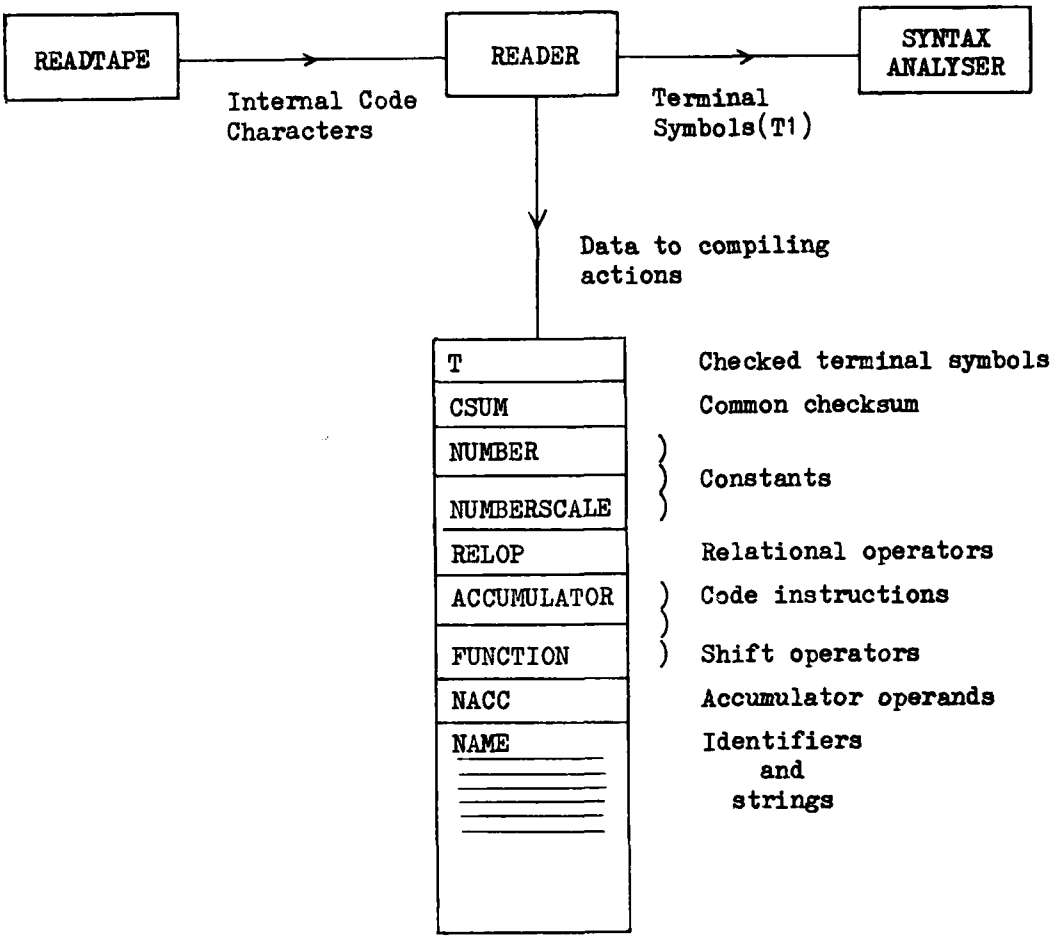
The Reader (pre-processor) is the only section of the compiler concerned with the individual characters which form the source program. It processes these, grouping them as required, to produce a series of symbols representing the source program. This symbol sequence is then analysed by the syntax analyser, which in this implementation not only checks for the legality of their occurrence within the sequence, but also invokes the required compiling actions at the appropriate points.

Some symbols may have an arbitrarily large number of character combinations allowable as their representation. These differing representations although syntactically indistinguishable are semantically distinct. Thus a symbol group may not only have a syntactic "value" associated with it, but may also have a semantic "value" which is a function of the actual characters themselves. For example: in this implementation, a non-zero integer constant has a single syntactic "value", but has over sixteen million possible semantic values (which in this case is its numeric value), and several times this number of permitted representations.

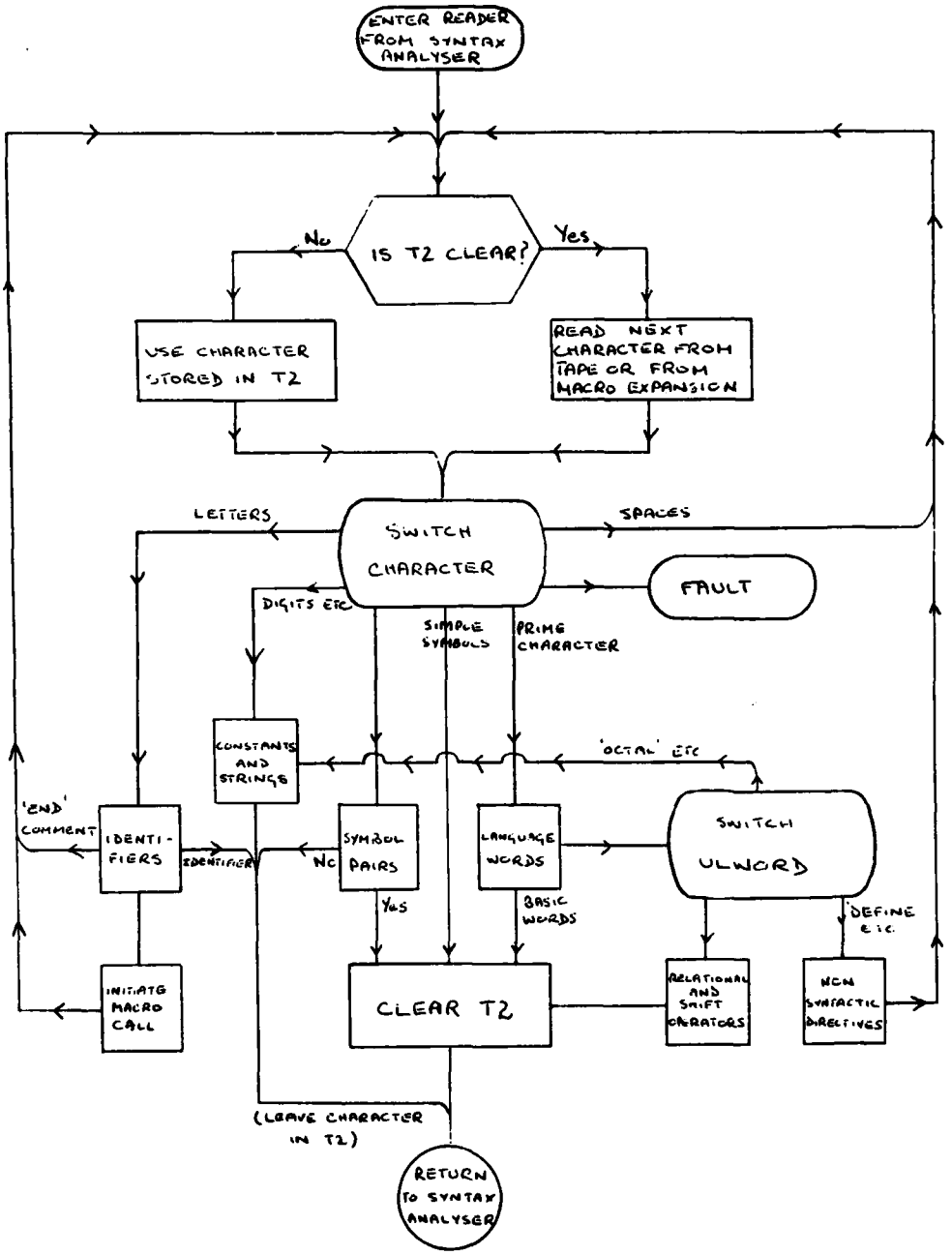
The syntactic symbols are passed to the syntax analyser, one symbol per call of READER, using the variable T1. As semantic values are evaluated, they are passed to the compiling actions through global variables, these being: NUMBER,NUMBERSCALE,RELOP,ACCUMULATOR,FUNCTION,NACC and NAME. In the last case NAME is the start of an area into which completed strings representing identifiers are placed. It would be possible to incorporate the syntax for identifiers into the table which is used to steer the syntax analyser. This would however prove to be rather slower in operation. As the macro expansion facility is required, the Reader must examine each identifier before it is "seen" by the syntax analyser, to determine whether it is the identifier of a macro. If macro expansion is not required, the string is moved to NAME. An alternative method could be used, where each identifier is represented by a unique integer value, new values being assigned to identifiers on their first occurrence. This technique is often used in conjunction with "Hash Tables".

The Reader removes from the source program all layout and comment, Macro definitions, deletions, and calls; and also 'HALT' directives and optional items.

READER: INFORMATION FLOW



READER: SIMPLIFIED FLOWCHART



LIST OF TERMINAL SYMBOLS PASSED TO SYNTAX ANALYSER

Value	Symbol	Representation
0	Zero Constant	Any zero constant
1	Integer Constant	Contains integer part only
2	Real Constant	Includes . and/or &
3	Shift Operator	'SRA', 'SLA', 'SRL', 'SLC'
4	Relational Operator	> 'GT', <= 'LE', <> 'NE', = 'EQ', < 'LT', >= 'GE'
5	String	Enclosed in " quotes
6	Code Instruction	Octal digit followed by 3 Letters
7	Differ	'DIFFER'
8	End	'END' , 'E'
9	If	'IF'
10	Colon	:
11	Semicolon	;
12	Then	'THEN'
13	Overflow	'OVERFLOW' , 'OVR'
14	No	'NO'
15	Trace	'TRACE'
16	Assignment	'ASSIGNMENT'
17	Loop	'LOOP'
18	And	'AND'
19	Or	'OR'
20	Union	'UNION'
21	Location	'LOCATION' , 'L'
22	Mask	'MASK'
23	Special	'SPECIAL'
24	Open Round Bracket	({[}
25	Close Round Bracket	) {]}
26	Multiply	*
27	Plus	+
28	Comma	, {'BIT'}
29	Minus	-
30	Self	'SELF', *
31	Divide	/
32	Accumulator	@0,@1,@2,@3,@4,@5,@6,@7
33	Array	'ARRAY' , 'A'
34	Begin	'BEGIN' , 'B'
35	Code	'CODE'
36	Do	'DO'
37	Else	'ELSE'
38	For	'FOR'
39	Goto	'GOTO' , 'G'
40	Common	'COMMON'
41	Integer	'INTEGER' , 'I'
42	Fixed	'FIXED' , 'F'
43	Floating	'FLOATING'
44	Label	'LABEL'
45	Procedure	'PROCEDURE' , 'P'
46	Identifier	Starts with Letter , £ , or %.
47	Unsigned	'UNSIGNED' , 'U'
48	Finish	'FINISH'
49	Switch	'SWITCH' , 'S'
50	Library	'LIBRARY'
51	Step	'STEP' , :
52	Table	'TABLE' , 'T'

53	Until	'UNTIL' , :
54	Value	'VALUE' , 'V'
55	While	'WHILE'
56	External	'EXTERNAL'
57	Absolute	'ABSOLUTE'
58	Overlay	'OVERLAY'
59	Open Square Bracket	[
60	Answer	'ANSWER'
61	Close Square Bracket	]
62	Power	↑ , **
63	Becomes	← , :=
64	With	'WITH'
65	Compile (Spare)	'COMPILE'
66	Bits (Bit)	'BITS'
67	Load (Spare)	'LOAD'
68	Dump (Spare)	'DUMP'
69	Recover (Spare)	'RECOVER'
70	Level	'LEVEL'
71	Test	'TEST'

{These alternatives may only be used in a limited number of cases}

Operation

On entry to the Reader, from the syntax analyser, the previous "terminal Symbol" is moved from T1 to T, this symbol having been checked and found to be legal. If the one character lookahead facility has been used, an unprocessed character will be waiting in T2. This, for example, might be the character following an identifier, which will not be a letter or digit. If no character is awaiting processing, a call is made of the procedure READ to supply the next character. This may come directly from the program source, or from a macro expansion.

Having obtained the next character to be processed, it is used as an index of the switch CHARACTER. This causes the section appropriate to the character to be entered. This switch has 64 entries, but as many of the entries are repeated it has only 18 distinct ways. This could probably be programmed in rather less than 64 code instructions, but would be slower, and much less amenable to alteration.

Simple Symbols

The characters ( ) + , - / [ ] ↑ ← are terminal symbols in their own right. The corresponding entries in CHARACTER cause a jump to be made to the label OK which is one of the exit points of the Reader. The character value has been assigned to T1, and T2 is cleared (set to -1) before a jump is made back to the syntax analyser at its label EXIT. This label is the common return point for all compiling actions, which although taking the form of a labelled block or section, are treated in the syntax as if they were procedures. This saves over four words per action.

The other exit point from the reader is EX. This is used where the character look ahead procedure READT2 has been used, and the character in T2 inspected but unprocessed. As the actual instruction labelled EX is a 'GOTO' EXIT all the jumps to EX will be set directly to EXIT, this being an effect of STATUS optimisation.

Language Words

A commonly occurring character is the ' character denoting the start of a "language word". As each of these words has a fixed meaning, they form in effect an extension to the character set. This is reflected in the operation of the section PR, which is entered in this case. After the word has been recognised by the use of RECOG, the value returned (range 7 - 90) is used as an index of the switch ULWORD, which is used in a manner similar to that of CHARACTER. As however values less than 72 represent language words which are terminal symbols in their own right, this case is treated separately by a test and jump to OK. This reduces the size of ULWORD to 19 entries, although it has only 11 distinct ways.

List of Recognised Language Words
(Lower case represents optional letters)

Word	Value
A (ARRAY)	33
ARray	33
ABsolute	57
AND	18

SWitch	49
SLA	79*
SLC	81*
SPecial	23
SRA	78*
SRL	80*
T (TABLE)	52
TEst	71
THen	12
TAble	52
TRace	15
U (UNSIGNED)	47
UNIon	20
UNTil	53
UNSigned	47
Value	54
WIth	64
WHile	55

\+ Value is not passed to syntax analyser

* Different value is passed to syntax analyser

? Spare word for future use

Character Pairs

Certain characters may be terminal symbols in their own right, or they may be the first of a pair of characters representing a terminal symbol. Each of these characters is allocated a section which tests the following character to determine if the two characters form a pair. These pairs are tabulated below:

Character	Section	Possible Pairs	Alternative
:	CN	:=	←
<	LS	<=	'LE'
		<>	'NE'
>	GS	>=	'GE'
*	AS	**	↑

If a pair is detected, a jump is made to OK to delete the second character, the appropriate value having been assigned to T1. If a pair is not detected, a jump is made to EX leaving the second character set up in T2 until the next entry to the Reader. If there were more pairs than tabulated above, it would be preferable to use a single section to process character pairs, using a preset table of combinations.

Comparators

The six possible comparators are treated as identical symbols syntactically. As it does not affect the syntax for comparison which comparator is used, they are all assigned the same terminal symbol value of 4. The variable RELOP is however set to one of six values depending on which comparator has been used.

There are two alternative representations for each comparator, either a single character or character pair (see above), or a two letter "language word". In the first case the sections GS, ES, & LS are entered via CHARACTER, and in the second case the section RO is entered via ULWORD.

The representations of the comparators, and the associated value of RELOP, are as follows:

Representation	Alternative	RELOP
>	'GT'	-2
<=	'LE'	-1
<>	'NE'	0
=	'EQ'	1
<	'LT'	2
>=	'GE'	3

These values have been carefully chosen. The two negative values represent comparators for which there is no equivalent machine instruction, but by reversing the order of the expressions to be compared (and adding 4 to RELOP) use may be made of available machine instructions. In the case of the four non negative values, the appropriate machine function code is obtained by adding octal twenty. This gives an if-false-jump instruction, whose sense may be reversed as required.

Shift Operators

The four shift operators are treated as identical symbols syntactically, and are assigned the terminal symbol value of 3. These are recognised by RECOG and the section SH entered via ULWORD. This calculates the function code number from the value assigned to the symbol, which is stored in T1. This number is then stored in FUNCTION.

Shift Operator	FUNCTION (octal)
'SRA'	30
'SLA'	31
'SRL'	32
'SLC'	33

Spaces and Newlines

Spaces are ignored in two ways. If a space is read and used as an index to CHARACTER, the entry is SP which causes the next character to be read. Where READT2 is used, this procedure specifically tests for the space character, and if it is found, reads the next character. Newlines are removed by READ.

Comments

Three types of comment are allowed, end comment, bracketed comment, and explicit comment.

After reading and packing an identifier, a check is made to see if the last terminal symbol was 'END'. This is performed by testing whether T contains the terminal symbol number for 'END', which is 8. If it does, the identifier is discarded and a jump back made to process the following character, which will be held in T2.

After each unquoted semicolon, occurring as a terminal symbol, or as the terminator of a 'DEFINE', 'DELETE', 'COMMENT' or 'PAGE' item, a call is made of SEMICOM.

This procedure checks for, and reads if present, bracketed comment. It keeps a count of the bracket level, so that matched round brackets are allowed within this type of comment.

Explicit comment, prefixed by 'COMMENT' (or 'C') is read in and ignored, up to and including the terminating semicolon. 'PAGE' comment is a special case of this type of comment. It is treated in the same manner, but where LEVEL is non zero, it is printed on the monitor printer.

Optional Items

The use of the unquoted ? character allows a limited form of compile time selection from the source program. If handswitch 1 is up, the character is ignored, and the following characters processed. If handswitch 1 is down, the ? is treated as if it were 'COMMENT', and the following characters up to and including the following semicolon ignored.

Code Instructions

Except in identifiers, and hexadecimal constants, a letter following a digit can only occur in code. This allows the fields of a code instruction to be used without intervening separators such as commas. Where an octal digit is followed by a letter, the digit is stored in ACCUMULATOR and a call made to RECOG to process the following three letter function code, the value returned being stored in FUNCTION. The flexibility of RECOG would allow function mnemonics to be of varying length, but the three letter groups are used to ensure as near as possible conformity with Astral. An accumulator-function pair has a terminal symbol value of 6.

Where an accumulator is to be used as an operand or modifier it is represented by the symbol @ followed by a single digit accumulator number. This has AMARK added and is stored in NACC. This representation has a terminal symbol value of 32.

Strings

Strings are processed by the section ST which is entered when the opening " is read. The string is assembled in MCID and subsequently moved to NAME, from where it is accessed by the compiling actions. As the string is terminated by another " it is necessary to have an alternative representation for this character within a string. This is achieved by representing the symbol as "". Thus to represent a string containing A"B it is necessary to write "A""B". A string is packed in standard Argus form and may contain up to 63 characters.

A string has a Terminal Symbol value of 5.

Constants

The Reader processes all constants completely, before passing a Terminal Symbol code representing the type of constant to the syntax analyser. Constants may be in one of four forms; Character, Hexadecimal, Octal and Decimal. In the first two cases the constant is an Integer, but in the last two it may be either Integer or Real. A zero constant, whatever its form, is treated as a special case. The constant is assembled in NUMBER and its scale factor is placed in NUMBERSCALE for subsequent use by the compiling actions. The types of constant and their Terminal Symbol values are tabulated below.

Type	Terminal Symbol value (T1)
Zero	0
Integer	1
Real	2

Character Constants

The official definition only specifies one form of character constant. This takes the form:

'LITERAL'(c)

where c is any printing character. In this implementation this may be abbreviated to $c. In either of these cases the standard internal character value is stored in NUMBER and the constant treated as an integer constant.

As a further extension, up to four characters may be used. These are packed right justified in a similar manner to Alpha constants in Astral. In this case the constant takes the form:

'LITERAL'n(ccc)

where n is in the range 1 - 4 and specifies the number of characters ccc. 'LITERAL' may be abbreviated to 'LIT'. These constants are processed by the sections CH ($), and LI ('LIT').

Hexadecimal Constants

These constants have only one form of representation, the symbol 'HEX' followed by one or more hexadecimal digits (0-9, A-F). These are assembled by the section HX as a right justified integer constant in NUMBER. A hexadecimal constant is treated as an Integer constant.

Octal Constants

These constants may be represented in a number of ways.

1 # ddd
2 'OCTAL' ddd
3 'OCTAL'(ddd)
4 #J ddd
5 'OCTAL' J ddd
6 #ddd.ddd
7 'OCTAL' ddd.ddd
8 'OCTAL'(ddd.ddd)

where ddd represents from one to eight octal digits, and 'OCTAL' may be abbreviated to 'OCT'. Forms 1, 2 & 3 are treated as right justified Integer constants,and forms 4 & 5 as left justified Integer constants. The remaining forms 6, 7 & 8 are treated as Real constants, and in this case the integer part must be less than #40000000. The Official Definition only permits forms 3 & 8.

Decimal Constants

These constants may be represented in a number of ways.

1 ddd

2 ddd.dd

3 &sd

4 ddd&sd

5 ddd.dd&sd

Where:

ddd represents 1 - 7 decimal digits whose value is less than 8388608

dd represents 1 - 6 decimal digits

sd represents a single decimal digit, optionally preceded by a sign (+,-).

The first form is treated as an Integer, the remaining forms as Real. The character & replaces the symbol 10 and is used to indicate powers of ten. The real and fractional parts of a Real constant are stored as a fixed point mixed number in NUMBER.FRAC prior to normalising to discover a suitable scale factor. Thus if a small number is represented in form (2), as a zero integer part and a fraction with leading zeros, accuracy will be lost. In this case it is better to use form (5), with a negative decimal exponent. eg 1.2&-3 will give a more accurate constant than 0.001200.

Macro Definition

Macro definitions are processed by the section DF which is entered when the symbol 'DEFINE' is recognised. This reads the macro definition and stores it in the area of the compiler's stack reserved for macros. The record of the macro definition consists of the following information stored in sequential locations.

1 A chain pointer to the previously defined macros.

2 The number of parameters required.

3 The macro name, in standard string form.

4 The parameter identifiers (if any), in standard string form.

5 The body of the macro, stored as packed characters, four to a word. The character " is represented by "", and the body is terminated by ";

It should be noted that no attempt is made at this point to recognise the occurrence of the identifiers of the parameters within the macro body. Although the body of a macro is enclosed in quotes and is similar in appearance to a string, it is not stored as a standard string as this would restrict the number of characters in the body to 63. Advantage is taken of the fact that the representation of the quote symbol within a macro body, as in a string, is by a pair of quotes. A macro body being terminated by a single unpaired quote, followed by an arbitrary character, semicolon being used for convenience. Unlike a string, where paired quotes are reduced to a single quote whilst packing, paired quotes are reduced to a single quote whilst unpacking a macro body.

Macro Deletion

Macro deletions are processed by the section DL which is entered when the symbol 'DELETE' is recognised. If the macro to be deleted is on the top of the macro stack, and macro expansion is not in progress, the macro is deleted by unchaining its record and recovering the space occupied. Its record is thereby lost.

Otherwise the macro is deleted by setting the first word of its identifier string to zero, thereby rendering its record inaccessable. In this case the space occupied is not recovered, as a "garbage collector" is not incorporated. Thus to ensure that the macro stack does not overflow, the following rule should be observed:

> Delete all macros in the reverse order to their definition.

It should be noted that the scope of macros does not follow the block structure of the language, all macros being effectively global irrespective of the block level at which they are defined. It is therefore suggested that the following rule be observed:

> All macros defined within a program segment should be deleted at the end of the segment.

Identifiers and Macro Expansion

An identifier commences with a letter A-Z, a £, or a %; and is followed by letters and digits. The identifier is assembled in MCID by the section ID calling IDENT.

If the identifier follows the symbol 'END' it is ignored as comment. If the identifier is not the name of a currently defined macro, or of a current macro parameter, it is moved to NAME and a Terminal Symbol value of 46 placed in T1.

Otherwise a new level of macro expansion is set up. If the macro requires parameters, the identifier must be followed by the correct number of parameters, separated by commas and enclosed in round brackets. Each of these parameters is used to form the body of a temporary macro which is given the name of the appropriate parameter, which is obtained from the record of the macro being invoked. Thus no special test is required for the occurrence of a macro parameter within a macro body. If a macro requires no parameters (and this applies also to parameters within macros) it may be optionally followed by a ! character, which is ignored.

Once a macro expansion level has been set up, characters are read sequentially from the macro body until the terminating unpaired quote is read. This sequence can only be interrupted by the setting up of a further level of macro expansion. When the terminator of a macro body is read, the macro expansion level is reduced, which deletes any macros declared within that level. The input is resumed from the next lower level, at the point at which it was interrupted, after restoring T2, if relevant. This case will only occur where a parameterless macro, or parameter, is not followed by the ! character.

Example of Macro Expansion

Consider the following portion of a program:

```
MACK
'BEGIN' 'INTEGER' ALPHA;
'DEFINE' LDX(ACC,ADD)"ACC!LDXADD!";
'CODE' 'BEGIN' LDX(7,ALPHA);
```

After storing the definition of LDX and reading the following 'CODE' 'BEGIN', the identifier LDX would be recognised as being that of a macro. After increasing the level of macro expansion by stacking the current values of MCCHAIN,T2,MCSOURCE,MCBODY and MCLIST, temporary macros would be set up for the two parameters. This would be equivalent to reading:

```
'DEFINE' ACC"7";
'DEFINE' ADD"ALPHA";
```

The body of LDX would then be read as the current input, and when the identifier ACC was recognised as being that of a macro a further expansion level would be set up, and the ! discarded. At this point, the next character to be read would be 7. The state of the macro stack at this point is given in the diagram below.

After reading the 7 the level would be reduced and the LDX would be read. As this occurs after an octal digit it would be treated as a function mnemonic and not an identifier. The letters ADD which follow would be treated as an identifier, which would be discovered to be a macro, and replaced by ALPHA. The expansion level would then be reduced twice, and the semicolon read. This would terminate the identifier ALPHA, which not being a macro would be passed on.

It should be noted that the occurrence of second parameter ADD in the body of LDX is followed by an ! character. This may at first sight seem to be superfluous. If it were not present however, the recognition of the termination of the occurrence of ADD as an identifier would only be achieved after the semicolon had been read; by which time the macro expansion level would have been reduced to zero, and the temporary macro for ADD deleted !

In fact, had the semicolon not been present, and the next characters were say X+, the effective identifier would have been ADDX, for which no definition or declaration exists.

A general rule for macro parameters, which is an oversimplification of the facilities available, is:

> Name all macro parameters %1,%2,%3 etc, and in the body of the macro refer to them as %1!,%2!,%3! etc.

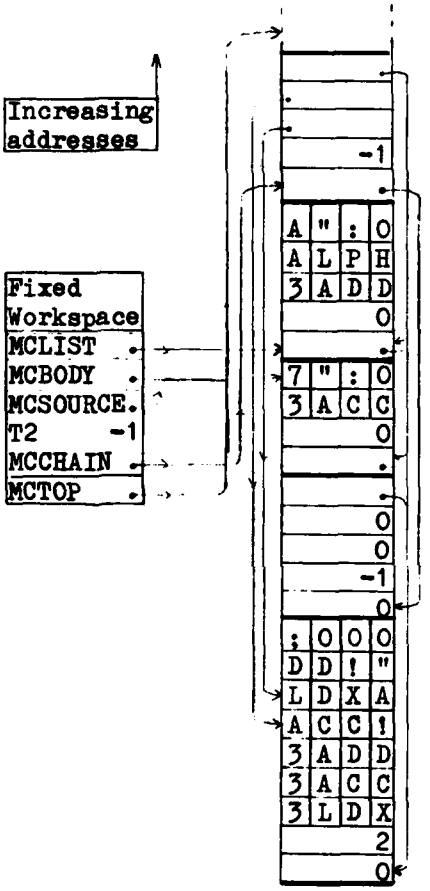
This rule is guaranteed to avoid confusion, and render macro definitions more legible.

Macro Stack state during expansion of macro

Input Tape: LDX(7,ALPHA);
↑

Level 1: ACC!LDXADD!
↑

Level 2: 7
↑ next character to be read



MCLIST MCBODY MCSOURCE T2 MCCHAIN	↑ Stacked workspace during replacement of ACC
Macro body ↕	↑ Definition of parameter ADD
Macro name Number of parameters Pointer to next macro	↓
Macro body ↕	↑ Definition of parameter ACC
Macro name Number of parameters Pointer to next macro	↓
MCLIST MCBODY MCSOURCE T2 MCCHAIN	↑ Stacked workspace during expansion of LDX
Macro body ↕	↑ Definition of macro LDX
Second parameter ↑ First parameter ↕ strings Macro name ↓	
Number of parameters Pointer to next macro	↓

Label AC:

This section is entered, via the switch CHARACTER, when the character @ is encountered. The following character must be an octal digit specifying an accumulator number for use in code. This number has AMARK added and is stored in NACC. A jump is then made to OK, leaving the value of T1 (32) unaltered.

'PROCEDURE' ADDCHAR;

This procedure is used to append the current character held in T1 to the standard string being formed in the MCID area at the base of the stack. The count field, in the first character position, is first incremented, and if it overflows (sets carry) a jump is made to FT. All identifiers are assembled in MCID in order that they may be tested to determine whether they are the identifiers of macros, or current macro parameters. If they are not, the string will subsequently be moved to NAME. For convenience, this process is also used for quoted strings, but in this case the completed string is not treated as a potential macro.

Label AM:

This section is entered, via the switch CHARACTER, when the character & is encountered. This denotes a Real constant which is the specified power of ten. NUMBER is set to 1024 and NUMBERSCALE to -10. This is equivalent to the constant 1, but guards against loss of significant accuracy. A jump is then made to EXPT to read the required power of ten.

Label AS:

This section is entered, via the switch CHARACTER, when the character * is encountered. The following character is read, using READT2, and if it is a second * , the * in T1 is replaced by ↑, and a jump made to OK. Otherwise a jump is made to EX.

Label CH:

This section is entered, via the switch CHARACTER, when the character ∅ is encountered. The following printing character is treated as an integer constant, and the value of its internal representation placed in NUMBER. A jump is then made to INTX via INTY, which reads the character following, into T2.

Label CM:

This section is entered, via the switch ULWORD, when the symbol 'COMMENT' (or 'C') is recognised. The following characters, up to and including the terminating semicolon, are read and ignored. A call is then made of SEMICOM to test for following bracketed comments, and a jump made to SP, to process the character left in T2 by SEMICOM.

This section is also used by PG (T1=83) to process page titles, and is slightly more complex than would otherwise be the case.

Label CN:

This section is entered, via the switch CHARACTER, when the character : is encountered. The following character is read, using READT2, and if it is the = character, the : in T1 is replaced by ← and a jump made to OK. Otherwise a jump is made to EX.

Label DD:

This section is entered, via the switch CHARACTER, when a digit is encountered. If this digit is followed by a letter, a call is made of RECOG, to process the three letter function code mnemonic, and return its number. The initial digit is stored in ACCUMULATOR, and the function number in FUNCTION. T1 is set to 6, the terminal symbol value for a code instruction, and a jump made to EX.

Otherwise a call is made of NOASSY to assemble the integer part of a decimal constant. If this is followed by the symbol for a decimal exponent (&), a jump is made to EXPT. Otherwise if the following symbol is not a decimal point, a jump is made to INTX. If a decimal point is present, a call is made of NOASSY to assemble the following fractional part, as an integer. This is then divided by the appropriate power of ten, to obtain an unsigned fraction which is stored in FRACTION. If the following symbol is not a decimal exponent symbol (&) a jump is made to NORM, otherwise the section EXPT is entered.

A flowchart covering the whole of the assembly of decimal numbers is given.

Label DF:

This section is entered, via the switch ULWORD, when the symbol 'DEFINE' is recognised. This section sets up the definition record of a new macro.

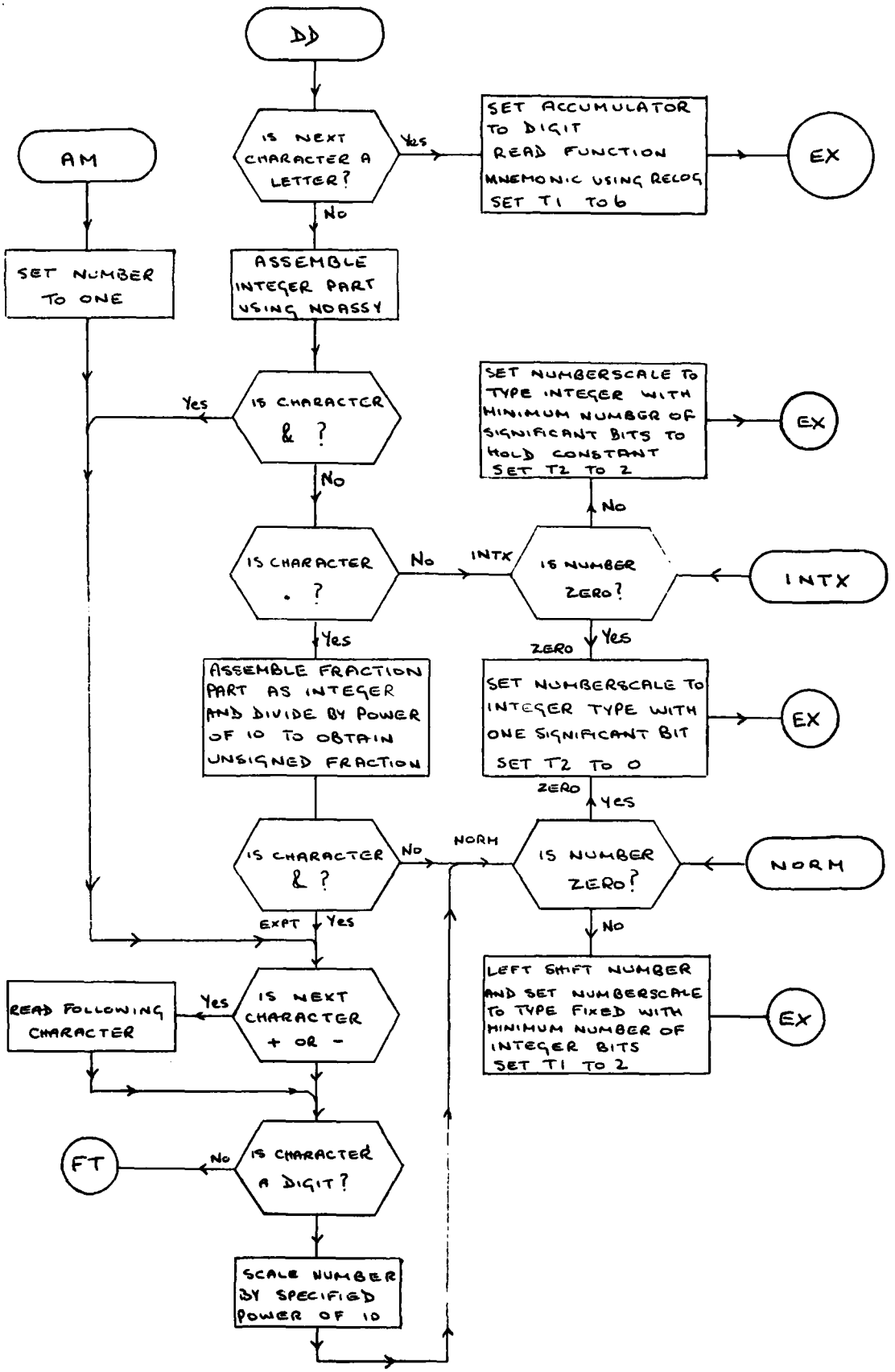
MCFLAG is set to indicate that a macro definition is in progress, and MCMAKE,MCIDENT and MCMOVE called to read the name of the macro and initiate its record. If this is followed by an open round bracket, the macro has parameters, and the names of these are read and added to the record. The parameter identifiers are separated by commas, and the list is terminated by a close round bracket.

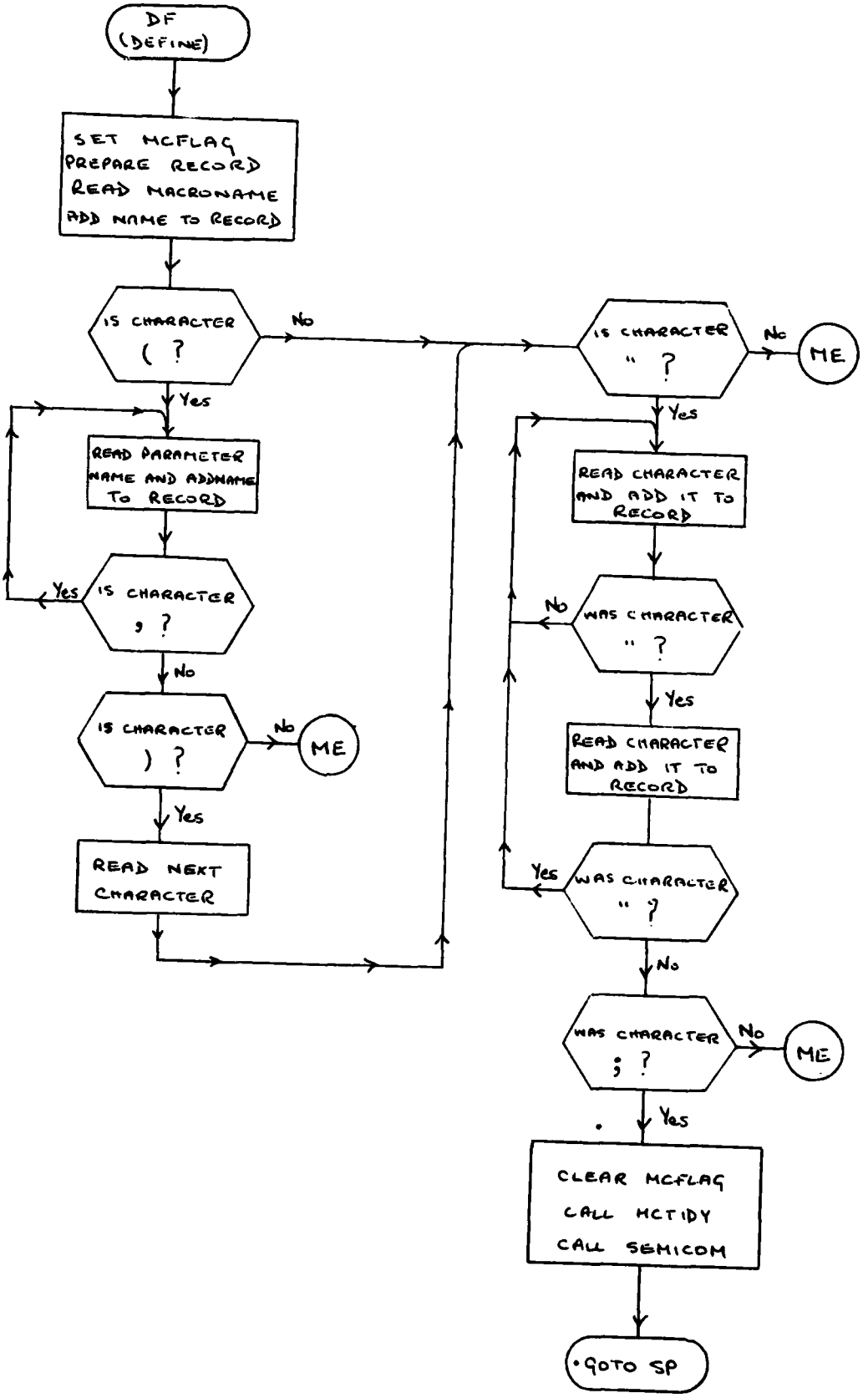
The macro body, which must start with a quote symbol, is read and added to the record, character by character, up to and including the last unpaired quote and the following mandatory semicolon. The macro definition is now complete, and MCFLAG is cleared. Calls are made of MCTIDY and SEMICOM. In the latter case this causes any following bracketed comment to be ignored. A jump is then made to SP to process the character left in T2 by SEMICOM.

A flowchart is given for this section.

Label DL:

This section is entered, via the switch ULWORD, when the symbol 'DELETE' is recognised. This must be followed by the identifier of a current macro, which is followed by a semicolon.


DD
IS NEXT CHARACTER A LETTER?
Yes
SET ACCUMULATOR TO DIGIT READ FUNCTION MNEMONIC USING RELOC SET T1 TO 6
EX
No
AM
SET NUMBER TO ONE
ASSEMBLE INTEGER PART USING NOASSY
IS CHARACTER & ?
Yes
SET NUMBERSCALE TO TYPE INTEGER WITH MINIMUM NUMBER OF SIGNIFICANT BITS TO HOLD CONSTANT SET T2 TO 2
EX
No
IS CHARACTER . ?
No
INTX
IS NUMBER ZERO?
INTX
Yes
ZERO
Yes
ASSEMBLE FRACTION PART AS INTEGER AND DIVIDE BY POWER OF 10 TO OBTAIN UNSIGNED FRACTION
SET NUMBERSCALE TO INTEGER TYPE WITH ONE SIGNIFICANT BIT SET T2 TO 0
EX
ZERO
Yes
IS CHARACTER & ?
No
NORM
IS NUMBER ZERO?
NORM
EXPT
Yes
No
LEFT SHIFT NUMBER AND SET NUMBERSCALE TO TYPE FIXED WITH MINIMUM NUMBER OF INTEGER BITS SET T1 TO 2
EX
READ FOLLOWING CHARACTER
Yes
IS NEXT CHARACTER + OR -
FT
No
IS CHARACTER A DIGIT?
SCALE NUMBER BY SPECIFIED POWER OF 10


DF
(DEFINE)
SET MCFLAG
PREPARE RECORD
READ MACRONAME
ADD NAME TO RECORD
IS CHARACTER
(?
No
Yes
READ PARAMETER
NAME AND ADDNAME
TO RECORD
Yes
IS CHARACTER
, ?
No
IS CHARACTER
) ?
No
ME
Yes
READ NEXT
CHARACTER
IS CHARACTER
" ?
No
ME
Yes
READ CHARACTER
AND ADD IT TO
RECORD
No
WAS CHARACTER
" ?
Yes
READ CHARACTER
AND ADD IT TO
RECORD
Yes
WAS CHARACTER
" ?
No
WAS CHARACTER
; ?
No
ME
Yes
CLEAR MCFLAG
CALL MCTIDY
CALL SEMICOM
•GOTO SP

Label DL/

The reference to the record of the macro is obtained using MCLOOK. If the macro does not exist, the reference will be zero, and this is treated as an error. If there are no macros currently being expanded and the macro to be deleted is the top macro on the stack, the macro is deleted by unchaining its record, and recovering the space occupied. Otherwise the macro is rendered anonymous by clearing the first word of its identifier string. The space occupied by its record is not recovered in this case. If however the macro is defined within a macro expansion, or is a temporary macro set up for a parameter, this space will subsequently be recovered. Thus as a general rule: delete macros in the reverse order to their definition.

Label ES:

This section is entered, via the switch CHARACTER, when the character = is encountered. T1 is set to 4, to indicate a relational operator, and RELOP set to 1. A jump is then made to OK.

Label EX:

This is the exit point from the reader in those cases where the character look ahead facility READT2 has been used, and an unprocessed character remains in T2. This will be processed on the next entry to reader.

Label EXPT:

This section is entered from AM and DD when the & character is encountered. This may optionally be followed by a sign (+,-), and then must be followed by a single decimal digit. The double length number NUMBER.FRAC is then repeatedly multiplied (+) or divided (-) by ten, the number of times being specified by the digit.

To improve the numerical accuracy, by preserving as many significant bits as possible, the calculation is carried out in a degenerate form of floating point arithmetic. To multiply by ten, the number is actually multiplied by 0.625 and 4 added to its exponent held in NUMBERSCALE. To divide by ten, the number is multiplied by 0.8 and 3 is subtracted from its exponent. Because of the use of multiplication in both cases, there is no possibility of overflow, and intermediate normalisation is not required.

The section NORM is then entered.

Label FI:

This section is entered, via the switch ULWORD, when the symbol 'FINISH' is recognised. T1 is set to 48, the value representing the symbol, and T2 is set to an arbitrary value selected from the set of characters which are terminal symbols in their own right. This ensures that no more tape is read after the syntax analyser has accepted the 'FINISH'. A jump is then made to EX which leaves T2 set.

Label FT:

This section is entered on the discovery of an illegal character, or combination of characters. Compilation is terminated with the message "CHARACTER FAULT".

'PROCEDURE' GENOCT;

This procedure processes the general case of octal numbers. These may have simply an integer part, in which case they are treated as Integer constants. Or they may have both an integer and a fraction part, in which case they are treated as real constants.

The integer part is first assembled by OCTALNO, and assigned to NUMBER. If this is not followed by an octal point, T1 is set to 1 to denote an Integer constant. Otherwise a check is made to ensure that the integer part appears positive, and T1 set to 2 to denote a Real constant. The fraction part is assembled by OCTALFRAC, the result assigned to FRAC, and NUMBERSCALE cleared.

Label GS:

This section is entered, via the switch CHARACTER, when the character > is encountered. T1 is set to 4, to indicate a relational operator, and the following character read, using READT2. If this is =,RELOP is set to 3 and a jump made to OK. Otherwise RELOP is set to -2, and a jump made to EX.

Label HA:

This section is entered, via the switch ULWORD, when a 'HALT' directive is recognised. A call is made of HALT with "DIRECTIVE" as the parameter. On return, READT2 is called to read the next character, and a jump made to SP to process it. This causes the 'HALT' directive to have no syntactic effect.

Label HX:

This section is entered, via the switch ULWORD, when the symbol 'HEX' is recognised. It assembles hexadecimal integers in NUMBER. After the last character, a jump is made to INTX.

Label ID:

This section is entered, via the switch CHARACTER, when a letter, or £, or %, occurring as the first character of an identifier is read. The remainder of the identifier is then read by IDENT.

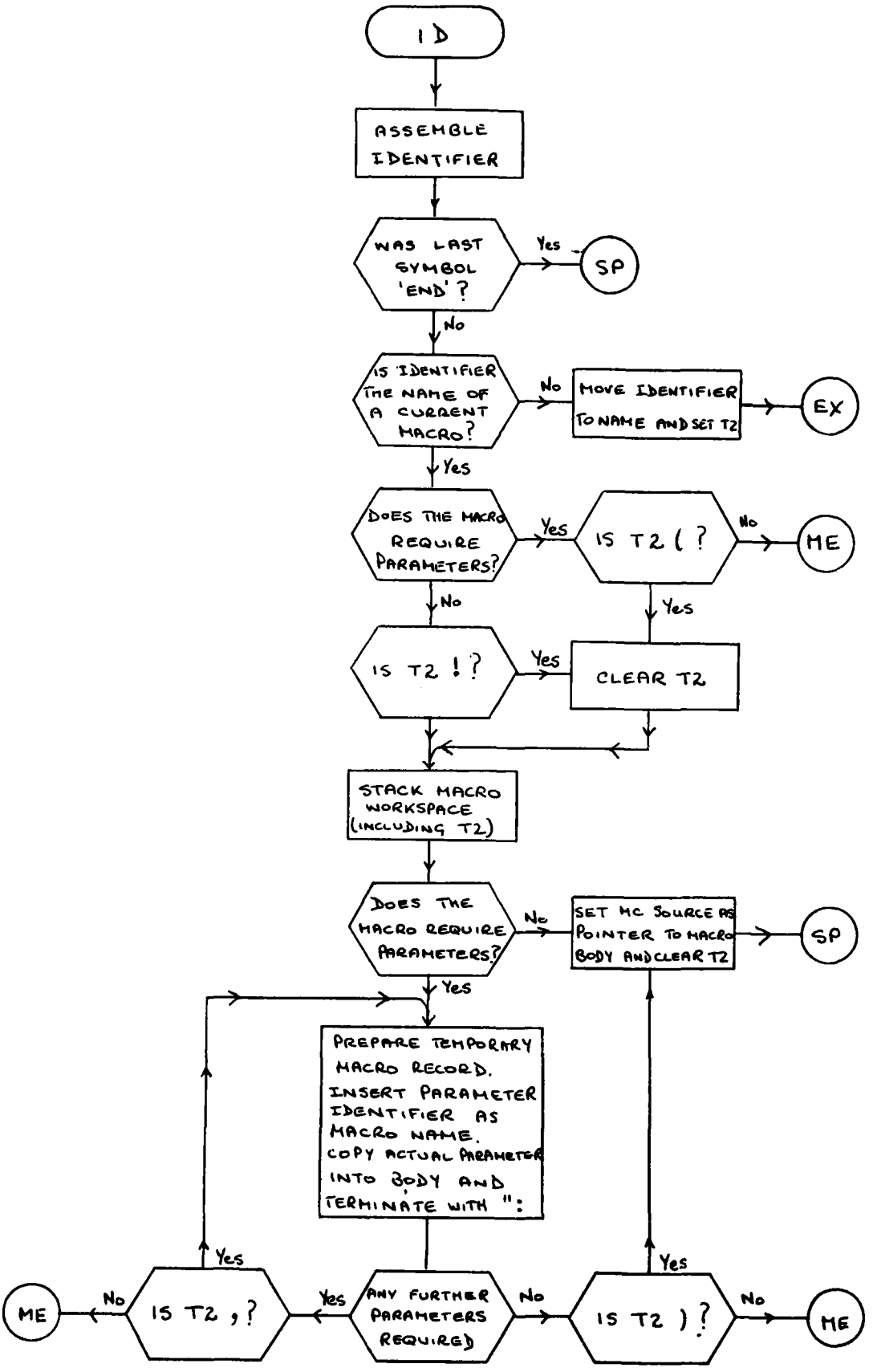
If this identifier follows the symbol 'END', it is ignored as this is an allowable form of comment. Otherwise MCLOOK is used to test whether the identifier is that of a current macro. If it is not, the identifier is MOVEd to NAME, the value (46) for an identifier assigned to T1, and a jump made to EX.

If the macro is parameterless, it may optionally be followed by ! , which is then ignored. (If this facility were not available it would be impossible to specify the accumulator field of a code instruction as a macro parameter.) If the macro requires parameters, they must be supplied, and in this case the macro identifier must be followed by an open round bracket.

After this check, a new macro expansion level is set up, by MOVEing the variables of the previous level onto the macro stack. A temporary macro is then set up for each parameter. These are parameterless macros having the parameter identifier, and their body consists of the actual parameter supplied by the call. These macros will be automatically deleted when the macro generation level is subsequently reduced at the end of the expansion.

The starting address of the macro body is set into MCSOURCE, and a jump made to SP to process the first character.

A flowchart of this section is given.


ID
ASSEMBLE IDENTIFIER
WAS LAST SYMBOL 'END'?
Yes
SP
No
IS IDENTIFIER THE NAME OF A CURRENT MACRO?
No
MOVE IDENTIFIER TO NAME AND SET T2
EX
Yes
DOES THE MACRO REQUIRE PARAMETERS?
Yes
IS T2 (?
No
ME
No
Yes
IS T2 ! ?
Yes
CLEAR T2
STACK MACRO WORKSPACE (INCLUDING T2)
DOES THE MACRO REQUIRE PARAMETERS?
No
SET MC SOURCE AS POINTER TO MACRO BODY AND CLEAR T2
SP
Yes
PREPARE TEMPORARY MACRO RECORD. INSERT PARAMETER IDENTIFIER AS MACRO NAME. COPY ACTUAL PARAMETER INTO BODY AND TERMINATE WITH ":
Yes
ANY FURTHER PARAMETERS REQUIRED
Yes
No
IS T2 , ?
No
ME
Yes
IS T2) ?
Yes
No
ME

'PROCEDURE' IDENT;

This procedure is used to assemble identifiers, after the first character has been checked. The identifiers are assembled into the area at the base of the stack, starting at MCID. This area is used as each identifier (except those following 'END', 'DEFINE' and 'DELETE') have to be treated as potential macro calls.

The area is first initialised as an empty string, and the first (checked) character inserted. Subsequent characters are read into T1 and also T2, and if the character is a letter or digit, it is added to the string by ADDCHAR. When a character is found which is not a letter or digit, this procedure is terminated, leaving the character in T2 for subsequent examination.

Label INTX:

This section is entered from DD,0C and INTY. In all these cases an integer, but not necessarily numeric, constant has been read. If this is zero, a jump is made to ZERO. Otherwise the integer is shifted left to discover the number of significant bits required to represent it without loss of accuracy. This number is required in certain cases of scaled arithmetic. T1 is set to 1 to indicate a non-zero integer constant, and NUMBERSCALE is set to indicate an integer constant requiring the number of significant bits determined above. A jump is then made to EX.

Label LI:

This section is entered, via the switch ULWORD, when the symbol 'LITERAL' is recognised. This is then followed by a single printing character, enclosed in round brackets. As an extension in this implementation, more than one character, but less than five, may be included between the brackets, provided that their number is specified by a single digit placed before the opening bracket. The characters are assembled as a right justified integer to base 64 in NUMBER, and a jump made to INTX via INTY which reads the following character into T2.

Label LS:

This section is entered, via the switch CHARACTER, when the character < is encountered. T1 is set to 4, to indicate a relational operator, and the following character read, using READT2. If this is =,RELOP is set to 01, or if it is >, RELOP is set to zero; in either case a jump is then made to OK. Otherwise RELOP is set to 2, and a jump made to EX.

'INTEGER' 'PROCEDURE' MCCHAR

('VALUE' 'INTEGER' CHAR);

This procedure is used to append the character CHAR to the body of the macro currently being defined. It uses MCTOP as a character modifier, (m.s. 2 bits give character position within word) which is incremented by one character position each time. The character CHAR is returned as the result in case subsequent examination is required.

This procedure is functionally the inverse of NEXTCHAR, but whereas the parameter of NEXTCHAR refers to an absolute core location, the implied parameter MCTOP, of MCCHAR refers to a core location relative to the start of the stack.

'PROCEDURE' MCIDENT;

This procedure is used to read and assemble the identifier of a macro which must follow 'DEFINE' and 'DELETE', and also the parameter identifiers in a macro definition. The first character is read into T1. If it is not a letter (A-Z), nor either of the characters £ or %, a jump is made to ME. Otherwise the identifier is read in using IDENT.

'INTEGER' 'PROCEDURE' MCLOOK;

This procedure is called to test whether the identifier held in the locations starting with MCID, is the identifier of a currently defined macro. If it is, a pointer to the macro definition record is returned, this value also being assigned to MAC. Otherwise a zero result is returned.

Using MAC as the control variable, with an initial value of MCLIST, a scan is made down the list of macros, comparing the macro identifier string with that at MCID, using TESTSTRING. The scan terminates either when equality is found, or the list is exhausted. The current value of MAC serves as the result in either case.

'PROCEDURE' MCMAKE;

This procedure is called to prepare for the definition of an additional macro. This may be a macro explicitly defined by the 'DEFINE' symbol, or a temporary macro defining the parameter of a macro as the actual parameter supplied.

Two locations on the top of the macro stack are reserved and initialised. The first is used as the macro chain, and is set to the current contents of MCLIST, which is then set to point at the new macro. The second word will be used to contain the number of parameters required by the macro, and is at this point set to -1. This two word record will be followed by the string giving the macro identifier, the parameter identifiers (if any), and finally the macro body.

'PROCEDURE' MCMOVE

('LOCATION' 'INTEGER' FROM);

This procedure is used during the definition of a macro, to allocate space for and to insert identifier strings into the head of the macro record. For each record, the first call is used to insert the macro name, and any subsequent calls insert the names of the parameters. The second word of the macro record has one added to it each time by this procedure, and as its initial value is -1, it may be seen that this location will contain a count of the number of parameters.

When setting up a record for an explicitly defined macro, the parameter FROM will be the location of MCID, which contains the current identifier. When setting up temporary macros defining the parameters of a macro, the parameter FROM will point at the identifier string of the parameter in the record of the called macro.

'PROCEDURE' MCTIDY;

This procedure is called to ensure that the macro stack pointer MCTOP, starts on a word boundary, (it is used as a character modifier during macro definition) and that its value does not exceed the size of the macro stack. If its value exceeds STACKSTART, the value for the starting point for declarations on the stack, compilation is terminated by a call of GIVEUP with "MACRO STACK OVERFLOW" as the parameter.

Label ME:

This section is entered when an error is discovered in the definition, deletion, or expansion of a macro. Compilation is terminated with the message "MACRO ERROR".

'PROCEDURE' MOVESTRING

('LOCATION' 'INTEGER' FROM,TO);

This procedure moves the string whose first location is given by the parameter FROM, to the area of core whose location is given by the parameter TO. The number of words to be moved is calculated from the count character of the string, this number being used as the first parameter of the call of MOVE which performs the operation.

'INTEGER' 'PROCEDURE' NOASSY

('VALUE' 'INTEGER' BASE,LIM);

This procedure is a general number assembly routine. It assembles up to LIM digits of a number to base BASE, where BASE may be in the range 2 to 10. On entry the first digit must be in T1, and the second character (which may not be a digit) must be in T2. The assembled number is returned as the result, and the number of digits processed is placed in NODS. If the first character is not a valid digit, or the number of digits exceeds the specified limit, a jump is made to FT.' This procedure is used to assemble both the integer and the fractional parts (as integers) of octal and decimal numbers.

Label NORM:

This section is entered from DD, EXPT, and OC. A double length number is supplied in NUMBER.FRAC with a binary scale factor in NUMBERSCALE. The number is shifted left as far as possible without overflow, NUMBERSCALE being adjusted according to the number of places of shift. Provided that the number is non zero, (if it is a jump is made to ZERO) the appropriate 'FIXED' scale is calculated for the number. This scale may subsequently be inserted in the TYPEBITS field of an Arithmetic Operand. The CON marker is set, the TYP field set to 1 (fixed), and the SGB field set to 24. The PVL field is calculated from the value of NUMBERSCALE. This gives a scale with the minimum number of integer bits required to hold the number. This is then stored in NUMBERSCALE.

Provided that this scale lies within the range allowed by the implementation, T1 is set to 2, to indicate a Real constant, and a jump made to EX.

Label OC:

This section is entered, via the switch ULWORD, when the character # is encountered. This denotes an octal constant. It is also entered at OQ and OX from OI.

If the following character is J, the number is a left justified integer. This is read using OCTALFRAC, which left justifies it, and a jump is made to INTX.

Otherwise a call is made to GENOCT to read an Integer or Real octal constant. If the constant is an integer, a jump is then made to INTX, otherwise a jump is made to NORM.

'INTEGER' 'PROCEDURE' OCTALFRAC;

This procedure returns a 24 bit (unsigned), left justified octal number. It is called after the octal point to assemble the following octal fraction, and after the symbols #J to assemble a left justified octal integer.

The first digit is read by a call of READT1, and the number assembled, right justified, by OCTALNO. This is then shifted left, the number of places of shift being calculated from the number of digits NODS, and returned as the result.

'INTEGER' 'PROCEDURE' OCTALNO;

This procedure is used to assemble octal numbers of up to eight digits. On entry the first digit must be in T1, and the next character is obtained by means of a call of READT2. NOASSY is then used to assemble the number.

Label OI:

This section is entered, via the switch ULWORD, when the symbol 'OCTAL' is recognised. In this implementation the following octal number may be unbracketed, if required, and in this case a jump is made to OQ. Otherwise, after the opening round bracket has been read, a call is made to GENOCT to read the number, and the closing round bracket checked. After reading the character following the closing brackets, a jump is made to OX. This redundant character read operation is required as an exit will subsequently be made via EX, which does not clear T2.

Label OK:

This is the exit point from the reader in those cases where the character held in T2 has been processed. T2 is cleared, by setting it negative, so that on the next entry to the reader a further character will be read.

Label PG:

This section is entered, via the switch ULWORD, when the symbol 'PAGE' is recognised. If LEVEL is non-zero, "PAGE:" is printed on a new line, and followed by the characters supplied by READ, up to but not including the terminating semicolon. This is the followed by two newlines. If LEVEL is zero, 'PAGE' is equivalent to 'COMMENT'.

Label PR:

This section is entered, via the switch CHARACTER, when the starting ' character of a "language word" is read. A call is made of READT2 to read the first letter, and then RECOG called, with a parameter of 1, to identify the symbol. The result of RECOG is assigned to T1, and if this is less than 72, it represents the value allocated to a terminal symbol, and a jump is made to OK. Otherwise the value is used to select an entry in the switch ULWORD.

Label QM:

This section is entered, via the switch CHARACTER, when the character ? is read. This is the starting symbol of an optional item. If Handswitch 1 is up (0), the ? symbol is ignored, and the following item processed in the normal manner. Otherwise the ? symbol is treated as the 'COMMENT' symbol, and the following characters, up to and including the terminating semicolon, ignored.

'INTEGER' 'PROCEDURE' READ;

This procedure is the source of characters for the pre-processor. They are obtained either from the input tape or from the expansion of a macro.

If there are no macros currently being expanded, READTAPE is called to read the next character from the input tape. If this character is the newline character, it is ignored, and the call repeated.

Otherwise, the next character is unpacked from the body of the current macro definition, using NEXTCHAR. If this character is the quotes character ("), the next character is also read. If this second character is also a quote, this is allowed to stand. Otherwise the macro expansion level is reduced by the use of MOVE to restore the "MC" variables of the next lower level, and to restore T2. Note that although these variables are only used in READER, they are declared in the global data, as their values are required to be preserved from one call of READER to the next. If a macro is currently being defined, MACFLAG will be set, and it is not possible to reduce the macro expansion level at this point. If the restored value of T2 is a valid character, this is allowed to stand as the next character. Otherwise a jump is made to the start of this procedure to obtain the next character.

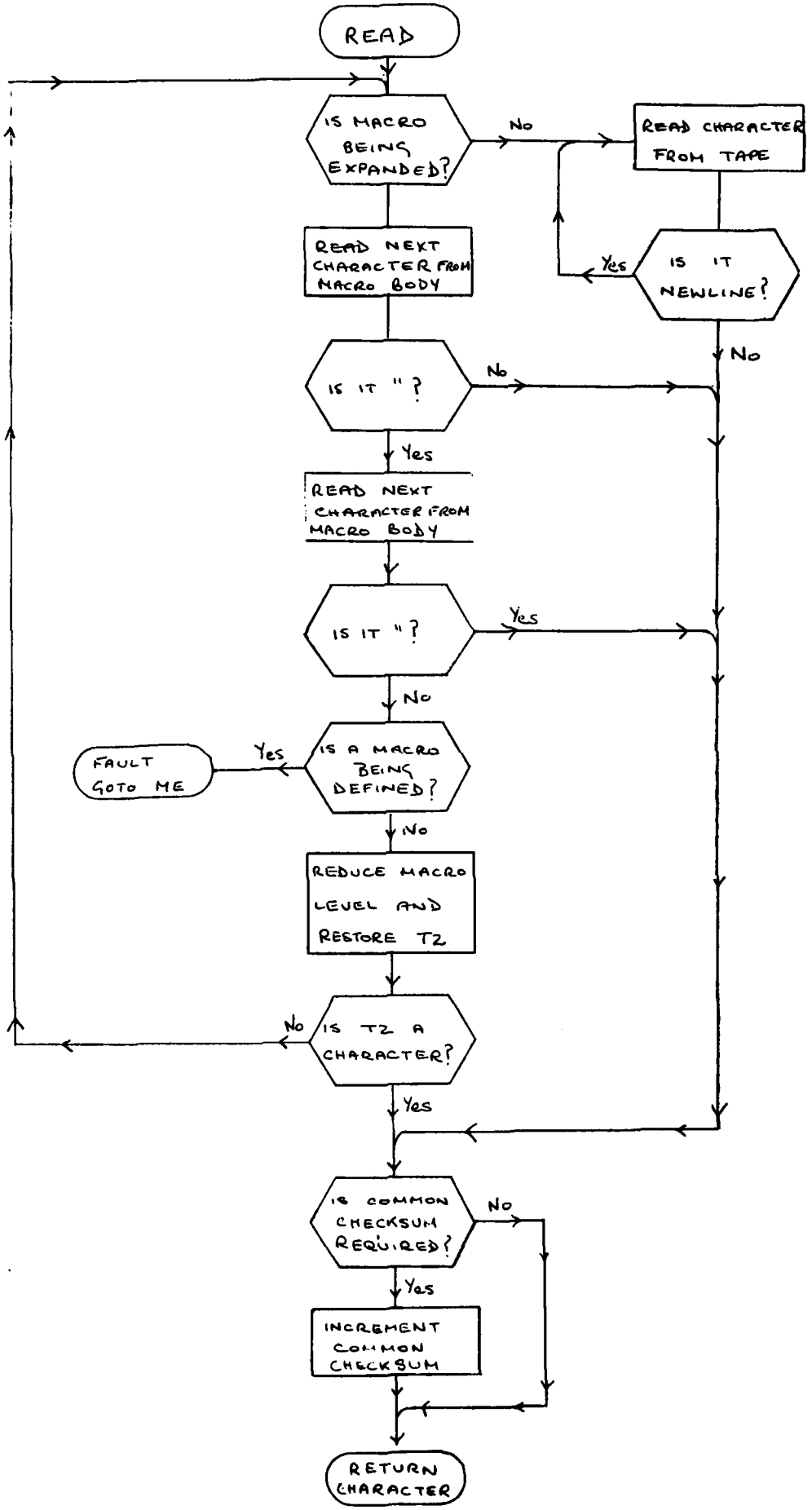
After the character has been obtained, it is used to form part of the common checksum CSUM if a 'COMMON' segment is currently being compiled.

A flowchart for this procedure is given.

'INTEGER' 'PROCEDURE' READT1;

This procedure is used as a one character look-ahead facility in cases where it is known that the following non-ignored character, if legal, must form part of a composite symbol. (e.g. a decimal point must be followed by at least one decimal digit.)

This procedure calls READT2 to obtain a character, which it stores in T1, and also returns as the result.


READ
IS MACRO BEING EXPANDED?
No
READ CHARACTER FROM TAPE
READ NEXT CHARACTER FROM MACRO BODY
Yes
IS IT NEWLINE?
No
IS IT " ?
No
Yes
READ NEXT CHARACTER FROM MACRO BODY
IS IT " ?
Yes
No
FAULT GOTO ME
Yes
IS A MACRO BEING DEFINED?
No
REDUCE MACRO LEVEL AND RESTORE T2
No
IS T2 A CHARACTER?
Yes
IS COMMON CHECKSUM REQUIRED?
No
Yes
INCREMENT COMMON CHECKSUM
RETURN CHARACTER

'INTEGER' 'PROCEDURE' READT2;

This procedure is chiefly used to give a one character look-ahead facility in cases where the next non-ignored character may, or may not, form part of composite symbol with the previous characters. (e.g. the character following two integer decimal digits may be another digit, a decimal point (.), or an exponent symbol (&); in which case it forms part of the number. Any other character does not.)

This procedure repeatedly calls READ until a non-space character is obtained. This is stored in T2 and also returned as the result. Unless overwritten, the character stored in T2 will be processed at the next call of READER.

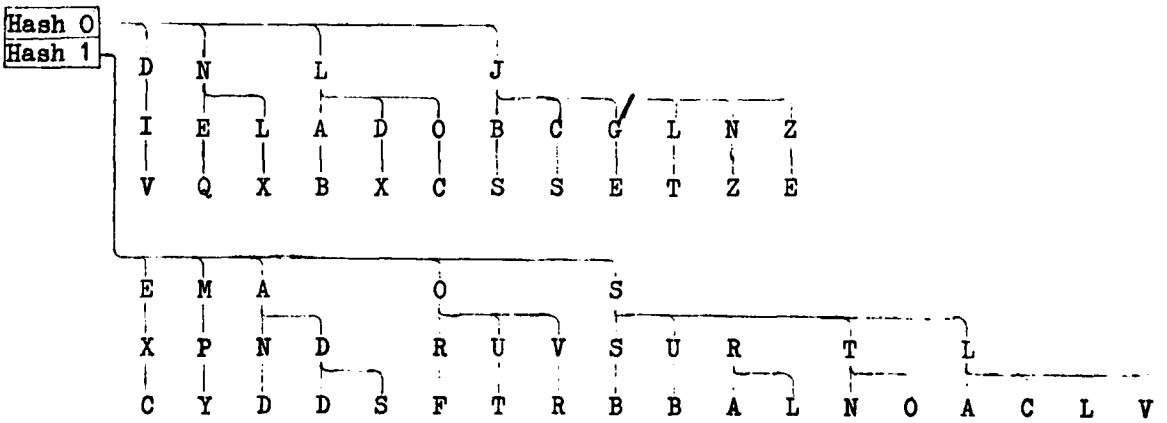
'INTEGER' 'PROCEDURE' RECOG

('VALUE' 'INTEGER' ULW);

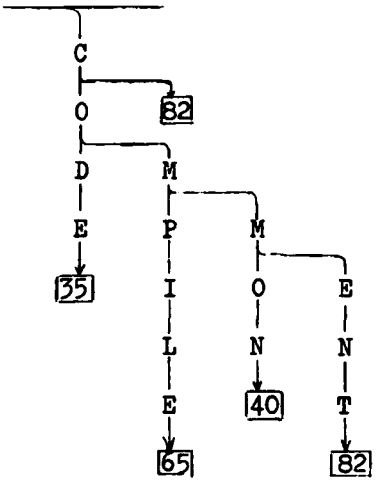
This procedure is a recogniser for alphabetic symbols which are members of two fixed groups. The first of these groups (ULW=1) being the larger group consisting of the "language words". (e.g. 'BEGIN' , 'END'.) In this second case the use of unambiguous abbreviations is allowed in all cases, and the use of a few specified ambiguous abbreviations is also permitted. (e.g. 'E' for 'END' (not 'ELSE')). Certain of these language words have other representations, which are not the concern of this procedure. (e.g. 'NE' and <>, 'STEP' and :).

This procedure operates by interpreting one of the two fixed arrays FUNCTIONS and ULWRDS. These consist of six bit characters, packed four to a word; and have been automatically generated from lists of the allowable symbols. Each letter in a symbol is represented by its internal code value (range 33 to 58, A to Z). Values in the range 60 to 63 represent terminating characters, and together with the following character give the number allotted to the symbol (0-255). Values less than 33 are "jump" characters, and are used in the selection of alternative routes.

These arrays are a one dimensional representation of a two dimensional list structure. This procedure moves through the structure, comparing the stored letters with the letters read in, and taking the appropriate action. Where differing symbols have a common starting letter or letters, no backtracking or scanning is required. In order to speed up the recognition process, and because of the limit on the "jump" size, each structure is entered via a small "hash" table stored at the start of the array. This is indexed by the least significant one (FUNCTIONS) or two (ULWRDS) bits of the first letter. The structure of FUNCTIONS is illustrated as follows:



The procedure moves across the diagram to select alternatives, and downwards to select successors. The diagram for ULWRDS is too large for simple representation. In this case the incoming symbol is terminated by a ' character, and if it is an unambiguous abbreviation, the rest of the route through the structure must not contain any branches. Part of the structure, of the entries for C, of ULWRDS is illustrated below:



On entry to the procedure, the array CHOOSE is used to select the array to be used, according to the value (0 or 1) of ULW. CHOOSE also contains the masks required to select the hashed entry into the selected table. The character address of the current character is stored in SI, which is used by the procedure READS, which uses *NEXTCHAR* to unpack successive characters.

On successful recognition of an alphabetic symbol, a result in the range 0 to 255 is returned. This is formed by the two least significant bits of the terminator (60 to 63) and the six bits of the following character. In all cases of unsuccessful recognition, a jump is made to the label FT.

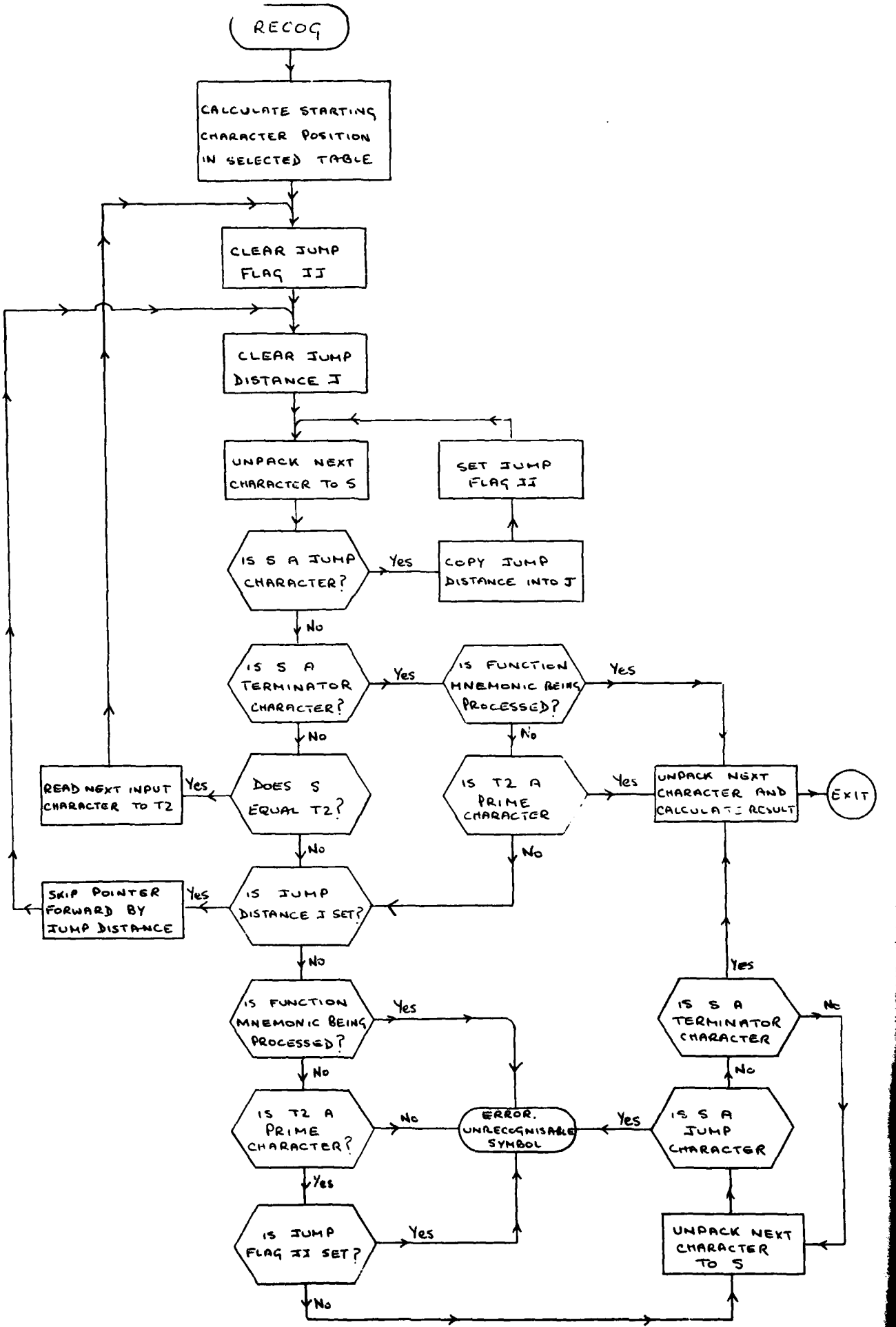
A detailed flowchart is given for this procedure.

Label RO:

This section is entered, via the switch ULWORD, when one of the relational operator symbols ('GT','LE','NE','EQ','LT' or 'GE') is recognised. The value assigned to RELOP is calculated from the symbol number, held in T1, which is set to 4. A jump is then made to OK.

Label SC:

This section is entered, via the switch CHARACTER, when a semicolon, occurring as a terminal symbol, is read. SEMICOM is called to process any following comment, and a jump is made to EX.


RECOG
CALCULATE STARTING CHARACTER POSITION IN SELECTED TABLE
CLEAR JUMP FLAG JJ
CLEAR JUMP DISTANCE J
UNPACK NEXT CHARACTER TO S
SET JUMP FLAG JJ
IS S A JUMP CHARACTER?
Yes
COPY JUMP DISTANCE INTO J
No
IS S A TERMINATOR CHARACTER?
Yes
IS FUNCTION MNEMONIC BEING PROCESSED?
Yes
No
No
READ NEXT INPUT CHARACTER TO T2
Yes
DOES S EQUAL T2?
IS T2 A PRIME CHARACTER
Yes
UNPACK NEXT CHARACTER AND CALCULATE RESULT
EXIT
No
No
SKIP POINTER FORWARD BY JUMP DISTANCE
Yes
IS JUMP DISTANCE J SET?
No
IS FUNCTION MNEMONIC BEING PROCESSED?
Yes
Yes
IS S A TERMINATOR CHARACTER
No
No
No
IS T2 A PRIME CHARACTER?
No
ERROR. UNRECOGNISABLE SYMBOL
Yes
IS S A JUMP CHARACTER
Yes
IS JUMP FLAG JJ SET?
Yes
UNPACK NEXT CHARACTER TO S
No

'PROCEDURE' SEMICOM;

This procedure is called to check for, and to skip over, bracketed comments following the semicolon symbol. One or more of these comments may follow the symbol, and these are detected by the outer 'FOR' loop, which reads the opening bracket, if present. The inner 'FOR' loop counts the bracket level B, as a side effect of the 'WHILE' element, and terminates when the matching closing bracket is found. On return from the procedure, T2 will contain the character following the last closing bracket.

Label SH:

This section is entered, via the switch ULWORD, when one of the shift operator symbols ('SRA','SLA','SRL' or 'SLC') is recognised. The required function code number is calculated, using the value associated with the symbol which is held in T1, and stored in FUNCTION. T1 is then set to 3, to indicate a shift operator, and the section OK entered.

Label SP:

This section is the starting point of the reader. If a valid character has been left in T2 by a previous call, then this character is copied into T1, otherwise READ is called and its result stored in T1. This character is then used to select an entry in the switch CHARACTER. This jumps to the section appropriate to the character. The characters may be categorised as follows:

1) Characters which are terminal symbols (e.g. +)

2) Characters which start terminal symbols (e.g. &)

3) Characters which may alone be terminal symbols, or may be part of a composite symbol (e.g. <)

4) Illegal Characters in this context (! and .)

5) Ignored characters (Space)

As the entry for Space causes this section to be re-entered, all Spaces supplied by READ are effectively ignored. It should be noted that T2 must never contain the code for Space.

Label ST:

This section is entered, via the switch CHARACTER, when an (opening) quote is read. The string is assembled, character by character, in the MCID area at the base of the stack using ADDCHAR, after this area has been initialised as an empty string. Paired quotes cause a single quote to be stored, and the string is terminated by a single unpaired quote. The string is then moved from the MCID area to the NAME area by MOVESTRING. The terminal symbol value of 5, the value for a string, is set into T1; and a jump made to EX, which does not delete the character in T2.

Label ZERO:

This section is entered from NORM and INTX when it is discovered that NUMBER is zero. For optimisation purposes a zero constant, whatever its form, is treated as a special case syntactically. T1 is set to zero to indicate this, and NUMERSCALE is set to indicate an integer constant requiring one significant bit only (sign bit). A jump is then made to EX.

COMPILER STRATEGY

Introduction

The strategy of the compiler is intimately tied up with the "working syntax" used to drive the syntax analyser. This syntax is based on the syntax of the Official Definition after it has been processed by Foster's SID (syntax improving device) program to reduce it to a "one track" form. This process introduces a number of auxiliary rules into the syntax, and these are named by appending one or more digits after the letter Q.

Into this syntax are embedded the names of the compiling actions required to be carried out at that point. The syntax has also been expanded in places to enable optimisations to be carried out. Some use has also been made of Simpson's "Semantically Selected" rules. In this case the alternative chosen at the start of the rule depends not on the incoming terminal symbol, but upon the result returned by a previous compiling action.

In each section of the strategic description the relevant part of the syntax will be quoted. The basic notation used is that of SAG, but with the following identifier conventions.

1	Rule Names	First letter in upper case, subsequent letters in lower case. The rule name is underlined where semantic selection is carried out.
2	Terminal Symbols	Lower case letters.
3	Compiling Actions	Upper case letters. The action name is underlined where it delivers a result to be used as a semantic selector.

Recapitulations will be enclosed in square brackets and dots used to denote alternatives irrelevant in the context.

Compiler Activation

```
Run=(Compileitem,Q70)

Q70=(finish)
    (semi,Run)
```

When the compiler is first loaded and entered, it initialises certain words of workspace, and then enters the syntax analyser. The first rule is named Run, and represents a list of items separated by semicolons and terminated by the symbol 'FINISH'. These are processed one by one, the compiler terminating awaiting re-activation after the last.

Segments

```
Compileitem=()
   (Program)
   (Commondec)
   (library,Q69)
   (external,orb,Extlist,crb)
   (absolute,orb,Abslist,crb)
   (Leveldec)
   (Yesno,Tracetype,trace)
   (test,SETTEST,Q68)
```

```
Q68=(Program)
   (Commondec)
   (library,CLEARTYPE,Q66)
```

Thus the list of items to be compiled may be split into three main categories:

1 Segments to be compiled

2 Communicators specifying identifiers external to a segment

3 Trace and diagnostic directives

Segments may be processed in a test mode by prefixing them with the symbol 'TEST'. This causes the action SETTEST to be executed which inhibits the paper tape output for the duration of the following segment.

Program Segments

```
Program=(id,BEGINPROG,begin,Body,ENDPROG,end)

Body=(D1,ENDDECS,S1)

D1=(STARTDEC,D,semi,Q50)

Q50=()
   (D1)
```

After the initial identifier giving the name of the segment, the action BEGINPROG is executed. This uses the identifier as part of the segment header output on tape. This must be followed by the symbol 'BEGIN', one or more declarations (each being terminated by a semicolon), one or more statements (separated by semicolons), and finally the symbol 'END'. Immediately before this final symbol the action ENDPROG is executed.

Before each declaration is processed the action STARTDEC is executed, and after the final declaration the action ENDDECS is executed. These are both involved with the housekeeping of storage allocation.

Declarations

```
D=(Type,Q47)
   (Tabledec,Presetlist)
   (switch,ADDRSW,TYPE SWITCH,Newid,becomes,Swlist)
   (Specialdec)
   (Overlaydec)
   (Valproc,Newid,BEGINPROC,colon,Newid,NEXTPSET,Procrest)
   (SETYES,TYPEPROC,Q48)
   (no,SETNO,Tracetype,trace)
   (Leveldec)
```

Thus a declaration list may consist of:

1 Data declarations, which may be preset in certain cases.

2 Overlayed data declarations which may not be preset.

3 Switch declarations.

4 Procedure declarations.

5 Trace and diagnostic directives.

It should be noted that at this point in the analysis, a declaration of a typed procedure cannot be distinguished from a simple data declaration, this being resolved in rule Q47. Similarly a declaration of an untyped procedure cannot be distinguished from a 'PROCEDURE' 'TRACE' directive, this being resolved in rule Q49 which is entered via rule Q48.

The declaration of each identifier causes an Identifier Specification Record to be generated, and placed on the compiler's main stack. This record is generated at the base of the stack, and consists of five words plus the string of the identifier. This information is built up incrementally using the overlayed names of the locations. Thus the information which will occupy the TYPEBITS word of the record on the stack is built up in the location named IDTYPE, chiefly by the actions named TYPE---. This information is usually complete by the time the identifier is read, and can be moved bodily onto the stack.

Data Types

```
[D1=(STARTDEC,D, . . .

    D=(Type,Q47)
       . . . . . .]

Type=(integer,Q59)
   (fixed,orbosb,int,NOBITS,comma,Sgint,NOAFTER,crbcsb)
   (floating,TYPEFLOAT)

Q59=(osb,int,NOBITS,csb,PARTINT)
   (TYPEINT)

Sgint=(int)
   (plus,int)
   (minus,int,NEGNUM)
```

The action STARTDEC clears IDTYPE prior to a number type being read. If the following symbol is 'FLOATING' the action TYPEFLOAT is executed, which sets the TYP field of IDTYPE to 2, the PVL field being left clear.

Alternatively if the following symbol is 'FIXED' this must then be followed by the specification of the number of significant bits, and the number of fractional bits. At the request of Ferranti this specification may be enclosed in either round or square brackets. (orbosb = (or [, crbcsb =) or]). After the integer constant specifying the number of significant bits, the action NOBITS is executed. This inserts the number of significant bits into the SGB field of IDTYPE. This is followed by a comma and a (signed) integer giving the number of fractional bits, which may be zero or even negative. (If the number is preceded by a - sign it is negated by NEGNUM). The action NOAFTER is then executed, which sets the TYP field of IDTYPE to 1, and calculates the scaling factor PVL.

The third possibility is the symbol 'INTEGER'. This may optionally be followed by a specification of the number of significant bits. This is only required where an integer is to be used in mixed arithmetic. In this case NOBITS is executed. This integer may only be enclosed in square brackets, as an ambiguity may arise concerning typed primaries if round brackets were allowed. After the closing bracket the action PARTINT is executed. This is in fact the same action as TYPEINT, which inserts the scale for integer into the PVL field of IDTYPE, (#67, P23) and leaves the TYP field set to zero.

The rule Type is used in many places in the syntax, and in all cases the relevant fields of *IDTYPE* will have been cleared before its use, often by the use of CLEARTYPE.

Simple Data Declarations

```
[D=(Type,Q47)
   . . . . . ]

Q47=(Q21,Presetlist)
   (TYPETPROC,procedure,Newid,BEGINPROC,Procrest)

Q21=(Idlist,DECSIZE)
   (array,TYPEARRAY,Arraylist)

Idlist=(Newid,Q52)

Q52=()
  (comma,Idlist)

Newid=(id,NEWNAME)
```

This set of rules states that a declaration of one or more simple variables consists of a data type followed by a list of identifiers, separated by commas.

The rule Type sets up IDTYPE and the action STARTDEC has previously inserted the current data address into DIRADD[0]. After each identifier, the action NEWNAME is executed. This checks that the identifier is unique at this block level, and moves the Identifier Specification Record onto the main stack. After this, it then increments DIRADD[0].

At the end of the list of identifiers, DECSIZE is executed, which copies the current value of DIRADD [0] back into DATAMAX, and hence allocates the storage for the variables.

Presetting Data Declarations

```
[Q47=(Q21,Presetlist)
     . . . . . .]

Presetlist=()
   (becomes,SKIPDTA,Presets)

Presets=(Presetnum,Q40)

Q40=()
   (comma,Presets)

Presetnum=(Pnumber,OUTPRESET)
   (orb,Presets,crb)
   (string,PRESETSTRING)

Pnumber=(pconst)
   (plus,pconst)
   (minus,pconst,NEGNUM)
   (ZERONUM)
```

Data declarations occurring at the head of a program may be preset. In this case the declaration is followed by a becomes (← , :=) symbol and a list of preset values. These are separated by commas and may be arbitrarily grouped within matched round brackets, to any depth. The number of items in the list must *not* exceed the number of items declared in the declaration.

When the becomes symbol is read, the action SKIPDTA is executed. This ensures that the loader will skip over any unpreset locations in the data area. The preset values are output by OUTPRESET after rescaling to the required scale. If this is not possible an error message is output. (Integers cannot be preset with real constants.) With the above restriction, any type of number, signed or unsigned, may be included in a list of presets. If a number is omitted (e.g. two commas occur in succession) ZERONUM is executed to give the effect of a zero constant.

As an extension, quoted strings may occur in the presetlist. In this case the variable is preset with the address of the string which is output by PRESETSTRING. This should only be used to preset integers. This gives the same effect as the assignment of a string.

Array Declarations

```
[D=(Type,Q47)
   . . .

Q47=(Q21,Presetlist)
    . . . .

Q21=(array,TYPEARRAY,Arraylist)
   . . . . . .]

Arraylist=(ZEROARRAYS,Idlist,osb,Boundpair,Morebounds,csb,ENDARRAY,Q42)

Q42=()
   (comma,Arraylist)

Boundpair=(Sgint,SETLB,colon,Sgint,SETUB)

Morebounds=(ONEDIM)
   (FIRSTDIM,comma,Boundpair,Arraytail)

Arraytail=(LASTDIM)
   (MIDDIM,comma,Boundpair,Arraytail)
```

When the symbol 'ARRAY' is read following a data type, the action TYPEARRAY is executed, which sets the AYM marker of IDTYPE. This ensures that the Identifier Specification Records of the following identifiers are flagged as being those of arrays, and will require subscripting in normal usage.

As a number of groups of arrays of the same data type but with differing dimensions and bounds may be declared within one declaration, each group is treated as virtually being a separate declaration by the rule Arraylist.

ZEROARRAYS is first executed, which clears the location ARRAYS used for counting numbers of arrays. (This is overlayed on DECLIST[0] and is incremented by NEWNAME.) The list of array identifiers is read by Idlist, and the Identifier Specification Records created, which do not at this stage have their address field correctly set, placed on the stack. The identifier(s) are then followed by a list, enclosed in square brackets, of pairs of integers

giving the bounds of each dimension of the array(s). These integers may be signed, and are separated by a colon. The second must not be less than the first. After the first integer SETLB is executed, and after the second of the pair, SETUB. These calculate the size and offset of the dimension.

If the array has only one dimension, ONEDIM is then executed. This inserts the addresses of the arrays into the Identifier Specification Records on the stack.

If the array has more than one dimension, the action FIRSTDIM is executed after the first pair of bounds. This inserts the addresses of the first level Iliffe vectors into the Identifier Specification Records. After each subsequent pair of bounds except the last, MIDDIM is executed. This outputs (to Special Data) the Iliffe vector refered to by the previous level, and referring to the next level. After the final pair of bounds LASTDIM is executed. This outputs the final Iliffe vector, which refers to the actual data addresses of the arrays.

At the end of a group of arrays ENDARRAY is executed. This uses the total data requirement (stored in ARRYS) to allocate data space.

Expressions

The compilation of expressions forms a very important part of the operation of the compiler, and probably accounts for the vast majority of the code generated. A very loose description of an expression is a collection of operands separated by operators. The syntax rules which follow express this in more detailed and rigorous terms. The priority of the operators expressed by these rules is as follows:

[Highest]	1	Shifts	'SRA',SLA',SRL','SLC'
	2	Logical AND	'MASK'
	3	Logical OR	'UNION'
	4	Non-Equivalence	'DIFFER'
	5	Exponentiation	↑, **
	6	Multiply/Divide	* , /
[Lowest]	7	Add/Subtract	+ , -

Within the compiler each of these operators has its associated compiling action. These operate on the two top Arithmetic Operand records on the stack, generate the appropriate instructions, and leave one Arithmetic Operand record on the Stack which gives the details of the result. These actions are invoked as soon as possible (bearing in mind the priorities of the operators), thus there is no reordering or regrouping of expressions. This is not very disadvantageous on a multi accumulator machine, and allows the programmer control over the order of evaluation.

Each expression is assigned an accumulator (PREFACC) in which the result is to be evaluated if possible. Further, if the scale to which the expression is to be evaluated is known, a marker (SCALEFIRM) is set, and the required scale placed in EXPSCALE. Otherwise the scale to which the expression will be evaluated depends only on the operands.

1 Simple Expressions

```
Expr=(Cexpr)
    (Axpr)
   (zero,CONSTANT,SCALETERM)
   (string,STRINGEX,SCALETERM)

Axpr=(Term,Q6)
   (Axpra)

Axpra=(plus,Term,Q6)
   (minus,Term,UNARYMINUS,Q6)

Q6=()
   (Q6a)

Q6a=(plus,Term,ADD,Q6)
   (minus,Term,SUB,Q6)
```

Ignoring, for the moment, the special cases, an expression consists of at least one Term, optionally preceded by a unary + or -, possibly followed by one or more terms, each being preceded by a + or - sign.

If the first term is preceded by a plus sign, this may be ignored. If however it is preceded by a minus sign, the Term requires negation, after evaluation. The action UNARYMINUS is executed in this case.

After each subsequent Term, the appropriate action ADD or SUB is executed. These output instructions for addition and subtraction.

2 Terms

```
Term=(Factor,Q5,SCALETERM)

Q5()
  (mult,Factor,MPY,Q5)
  (div,Factor,DIVIDE,Q5)
```

These rules give the next level of expression. It should be noted that the action SCALTERM is executed after each Term. In cases where no required scale is demanded, this action determines the scale to which the expression will be evaluated, after examining each term in turn. In cases where a scale is mandatory, its action is limited to the output of instructions for fixed to floating, and floating to fixed conversions. It is preferable that these conversions are performed as soon as possible, as they involve calls on Library procedures, and may require the temporary storage of intermediate results.

3 Factors

```
Factor=(Difference,Q24)

Q24=()
    (power,Difference,RAISE,Q24)
```

This rule introduces the exponentiation operation into this implementation. Care should be taken however, as unless the right hand operand is the integer constant 2, the result will be in floating point form, and may cause the rest of the expression to be evaluated by floating point arithmetic. The special case is treated as squaring, and is performed by multiplication.

4 Word Logic

```
Difference=(United,Q4)

Q4=()
   (differ,United,NEQ,Q4)

United=(Collation,Q3)

Q3=()
   (union,Collation,ORF,Q3)

Collation=(Tert,Q2)

Q2=(mask,Tert,MSK,Q2)
```

These rules govern the priority of word-logic operators. It should be noted that the results of these operations are regarded as being of type integer (of unspecified significance) regardless of the types of the operands, which strictly should be typed primaries.

5 Shifts

```
Tert=(Sec,Q1)
   (Pri,Q1)

Q1=()
   (shift,SETSHIFT,Shifts,DOSHIFT,Q1)

Shifts=(Sec,PLUSSUB)
   (Tpri,SCALETERM,PLUSSUB)
   (orb,Subex,crb)
```

Because of the nature of the shift functions, the right hand operand must be of type integer. Furthermore, if it is a variable or an expression, it is required to be held in a modifier before use. Thus it may be seen that the operand giving the number of places of shift has a similarity to a subscript, and in this implementation uses many compiling actions in common. The left hand operand may be of any type, and the result is regarded as being of type integer.

6 Word Slicing

```
Sec=(Bitset,BITSIN,Pri,RHSBITS)

Bitset=(bits,CLEARTYPE,osb,int,Oneormore,FIELDPOSN,UNSFIELD,csb)

Oneormore=(NOBITS,NOSIG,PARTINT,comma,int)
   (ONEBIT)
```

The compiling actions in the rules Bitset and Oneormore cause a data specification to be calculated in IDTYPE, and a part word specification to be calculated in BITSPEC, in a similar manner to those required for an unsigned integer table field. These values are stored by BITSIN for subsequent use by RHSBITS. This latter action does not usually involve the generation of any instructions at this point. Unless the PARTWORD field of the Arithmetic Operand of the Primary is non-zero, the two values are simply inserted into the TYPEBITS and PARTWORD fields of the record.

If instructions have to be generated at this point, it usually indicates bad programming practice, e.g. taking a part of a table field which is itself a part of a word. (This should be avoided by specifying a field consisting of the relevant bits.)

The result of this "operation" is of type integer, having the number of significant bits (plus one for the (zero) sign bit) given in the bit specification. Where the number of bits is one, this (and the following comma) may be omitted, thus allowing 'BIT' [9] etc.

7 Untyped Primaries

Primaries are the building blocks from which expressions are constructed, operators being the mortar. These may be divided into two main categories, typed and untyped. The untyped primaries do not have an intrinsic scale, this having to be determined by the compiler.

Within the compiler, the net result of processing a primary is to leave on the stack an Arithmetic Operand record giving type and address information for the primary. (This is usually also the case when processing the special cases of expression.)

```
Pri=(Tpri)
   (realcon,CONSTANT)
   (orb,STACKEXPR,SETPREF,Expr,crb,OFFSTEXPR)
```

In the case of a Real constant (ie containing . or &) there is no definite scale associated with the constant. A scale may however be chosen which has the minimum number of integer bits required to hold the number. This scale is supplied by READER, and is used by CONSTANT when setting up the Arithmetic Operand record. As constants are a special case of this type of record, the marker CON (m.s. bit) of TYPEBITS of the record is set to denote that the DIRADD field contains the actual constant and not an address. After any subsequent rescaling, when this record is used as an operand record for the output of an instruction, this constant is converted to an "address" by OUTWCONST.

In the case of an untyped bracketed expression, the scale to which it is evaluated is determined by the terms of the expression itself, following the rules for expressions occurring in an undefined context, irrespective of the scale of the expression of which it forms a part. (It should however be noted that a bracketed expression following a shift operator is implicitly typed Integer.) The action STACKEXPR is executed to set up a new "level" of expression evaluation. As the scale to which this level is to be evaluated is undefined, SCALEFIRM is cleared, and EXPSCALE initially set to zero. The preferred accumulator PREFACC, which will be used if possible for the evaluation of the expression, is set by SETPREF. After processing the following expression, the action OFFSTEXPR is executed, which reduces the expression level by restoring the variables concerned with the evaluation of expressions to their previous values, and also "floats" down the stack the Arithmetic Operand record left by the expression.

8 Typed Primaries

```
[Pri=(Tpri)
     . . . .]

Tpri=(intcon,CONSTANT)
   (id,LOOKUP,Q28)
   (NONAME,Singlesubs)
   (CLEARTYPE,Type,TYPEXPR,orb,Expr,crb,EXPRTYPE)
   (location,orb,Locvar,LOCACT,crb)
   (label,orb,id,Q32,crb)

Q28=(Subscript,VARCHECK)
   (RHSPTEST,Rhspsel)

Rhspsel=(VARCHECK)
   (Proccall)
```

The simplest case of a typed primary is a (non zero) integer constant. In this case CONSTANT is executed to set up an Arithmetic Operand record, as it is for real constants in untyped primaries, but in this case however, the scale supplied by READER is of integer typed, but with the number of significant bits specified in SGB.

An identifier occurring as a primary may be the identifier of either a simple variable, an array or table (field) name, or the name of a procedure. Before this is resolved, LOOKUP is executed. This action searches through the whole of the list of Identifier Specification records until one is found with a matching identifier string. The first five words of this record are then copied onto the top of the stack to form the Arithmetic Operand record, and the first word of this (SPIEL) is set to point at the identifier string on the stack.

If the identifier is followed by a subscript (starting with [ ) rule Q28 selects the first alternative. The actions associated with subscripts check that the identifier may validly be subscripted, and VARCHECK check that its type is Arithmetic. (It could have been a switch identifier). Otherwise the action RHSPTEST is executed. This returns a result of one if the identifier is that of a typed procedure, and zero otherwise. This value is used to select the appropriate alternative of Rhspsel. If the result supplied by RHSPTEST is zero, VARCHECK is executed to check the type of the identifier further (switch and untyped procedure illegal) and that no subscript is required (AYM clear). Otherwise, if the result is one, the rule Proccall is selected.

An anonymous reference consists of a single expression enclosed in square brackets. In this case NONAME is executed, which sets up an Arithmetic Operand record, with no name or address information, but of type integer array. The actions associated with subscripts will insert the address information, and delete the array marker, leaving the type set to integer.

Where a bracketed expression is preceded by a data type, the expression will be evaluated to the type specified. CLEARTYPE is executed before the type information is processed by Type, this clearing IDTYPE in readiness. The action TYPEXPR is then executed, which sets up a further expression level, in a similar manner to STACKEXPR, but setting SCALEFIRM to denote that EXPSCALE (which is set from IDTYPE) contains a mandatory scale. At the end of the expression, EXPRTYPE reduces the expression level in a similar manner to OFFSTEXPR, but only after ensuring that the expression has been evaluated to the required scale.

9 Address Primaries

```
[Tpri= . . . .
   (location,orb,Locvar,LOCACT,crb)
   (label,orb,id,Q32,crb)]

Locvar=(id,LOOKUP,Q29)
   (NONAME,Singlesubs)

Q29=(Subscript)
   (AYMCHECK)

Q32=(LOOKUP,Singlesubs,LABSK)
   (LABL)
```

The address of any item of data may be obtained by the use of 'LOCATION'. (In this implementation this includes special data, and in particular the locations of pointers to procedures, and of entries in switch lists.)

Although the type of the identifier is immaterial, it must be subscripted if a subscript is normally required, and unsubscripted otherwise. The actions associated with subscripts check the first case, and AYMCHECK the second. After this, the action LOCACT is executed. This changes the type of the Arithmetic Operand record to Integer, and except in certain optimised cases, sets the marker LCM to denote that the address itself is to be used as the operand instead of a reference to an operand.

As an extension, the address of a labelled statement may be obtained by the use of 'LABEL'. This must be followed by an opening round bracket and an identifier. At this point it is not yet determined whether the identifier is that of a label (which requires looking up in the label list) or that of a switch (which requires looking up in the main declaration list). This is resolved in rule Q32. In the case of a label, no subscript follows, and the action LABL is executed. This looks up the label in the (local) label list, and forms an Arithmetic Operand of type Integer for it. This has the marker LBM set to denote that the direct address contains a pointer to the label record (at this point the label may not be set), and the marker LCM also set. This reference to a label record will subsequently be converted to an "address" by USELAB.

If the identifier is subscripted, LOOPUP is executed, and the subscript processed by Singlesubs. The action LABSK is then executed, which checks that the Arithmatic Operand record is that of a switch, and changes its type to Integer. Note that LCM is not set, as the label address will be copied from the switch list.

```
('LABEL'(S[K]) ≡ ['LOCATION'(S[K])] !)
```

Special Cases of Expressions

```
[Expr= . . .
    (zero,CONSTANT,SCALETERM)
    (string,STRINGEX,SCALETERM)]
```

These two cases are chosen because, although they are similar to typed primaries, their explicit use in conjunction with an arithmetic operator is extremely obscure.

The detection of a zero constant as a special case enables useful optimisations to be performed in Assignments and Comparisons. In cases where these optimisations are not required zero is treated as an integer constant by CONSTANT, and its type (trivially) changed to that required by the context by SCALETERM.

The inclusion of strings is principally to allow them to be used as parameters of procedures, there being no operators which act on the individual characters of a string. The action STRINGEX outputs the string and sets up an Arithmetic Operand record of type integer, with the LCM set so that the address of the string is obtained.

Subscripts

```
Singlesubs=(Substart,csb,KILLEXPR)

Subscript=(Substart,Moresubs,csb,KILLEXPR)

Substart=(osb,SETUPSUB,Subex)

Moresubs=()
   (comma,SUBCOMMA,Subex,Moresubs)
```

Arrays may have more than one dimension, but switches and anonymous references are strictly one-dimensional. This is reflected in the first two rules above.

After the opening square bracket has been read, the action SETUPSUB is executed. This checks that the Arithmetic Operand record on top of the stack is flagged as requiring a subscript. A new level for expression evaluation is set up, requiring a mandatory integer scale, the preferred accumulator being a modifier.

After each comma separating subscript expressions, the action SUBCOMMA is executed. This outputs an instruction to copy an entry in an Iliffe vector into a modifier. In this way, any number of dimensions may be processed.

At the end of a subscript, the action KILLEXPR is executed to reduce the expression level. (Note that this discards any result of the expression, this will be explained below.)

```
Subex=(Subterms)
   (Cexpr,PLUSSUB)
   (Term,PLUSSUB,Subexcont)
   (zero)

Subexcont=()
   (Subterms)

Subterms=(plus,Term,PLUSSUB,Subexcont)
   (minus,Term,MINUSSUB,Subexcont)
```

A subscript expression is not evaluated as a normal expression, but is processed Term by Term by the actions PLUSSUB and MINUSSUB, which both use the procedure SUBTERM. This optimises subscript expressions containing only one integer variable and constants for a one dimensional non-parametric array, or only integer constants for a parametric or two dimensional array. As each Term is processed, its record is discarded, as the information resulting from the Term is stored in the address fields of the record of the operand being subscripted. Thus there is no Arithmetic Operand record left on the stack as the result of a subscript expression. Note that in the case of a subscript expression consisting of a zero constant, no action need be executed.

Conditional Expressions

```
[Expr=(Cexpr)
      . . . .]

Cexpr=(if,IFEX,Conditionlist,then,THENEX,Expr,else,ELSEFI,
            ELSEX,Expr,ELSEFI,FIEX)
```

As a conditional expression is an alternative case of expression it does not need to set up a further level for expressions, even where a conditional expression occurs within a conditional expression. A further level will however be temporarily set up for the evaluation of the conditions(s).

If the expression level at which the conditional expression is to be evaluated has SCALEFIRM set, i.e. a definite scale is required, both the consequence and alternative expressions will be evaluated to this scale. If however, it is not initially set, it will be set after the consequence expression has been evaluated, to ensure that the alternative expression is evaluated to the same scale. In either case however, the result of each expression will be evaluated in the accumulator specified by PREFACC.

After the symbol 'IF' has been read, the action IFEX is executed. This sets up a level for conditionals, and dumps any Arithmetic Operands existing only in accumulators. The following condition(list) is then processed, and the symbol 'THEN' read. The action THENEX is then executed, this being concerned with the output of a jump over the consequence expression, and setting the DUECHAIN if the symbol 'OR' occurred in the conditionlist.

The consequence expression is then processed, and ELSEFI executed after the symbol 'ELSE' is read. This action ensures that the result of the expression is in the accumulator specified by PREFACC. The action ELSEX is then executed, which outputs a jump over the alternative expression, sets the chain for the jump to the alternative expression, and sets SCALEFIRM. After the alternative expression has been processed, ELSEFI is again executed. The final action is FIEX. This reduces the conditional level, and creates an Arithmetic Operand record for the result of the conditional expression. This record will have its TYPEBITS set from EXPSCALE, and its direct address will be that of the accumulator specified by PREFACC.

Conditions

```
Conditionlist=(Condition,Q31)

Q31=()
   (and,OUTCJ,Conditionlist)
   (or,ORACT,Conditionlist)

Condition=(Yesno,overflow,OVRTEST)
   (STACKEXPR,ANYPREF,Lcond,Rcond,Relation)

Yesno=(no,SETNO)
   (SETYES,TYPEPROC)
```

In the interests of efficiency, it is required that conditions are evaluated only as far as is necessary to determine their truth or falsity. Thus, following the occurrence of the symbols 'AND' and 'OR', the associated actions output the appropriate jump instruction to test the preceding condition. This enables inconsequential tests to be skipped.

The above syntax does not need to express the relative priority of these two "Boolean operators", as this is obtained as a consequence of the operation of the tests output by the compiling actions.

Following the occurrence of the symbol 'AND', the action OUTCJ is executed. This outputs an if-false-jump instruction to skip over the following condition, or conditions if more 'AND's follow. Unless the symbol 'OR' intervenes, the jump will be made to the alternative, if any.

Following the occurrence of the symbol 'OR', the action ORACT is executed. This outputs an if-true-jump,directly to the consequence (statement or expression as appropriate). The destination of any preceding 'AND' jumps is then set to the following condition.

In this implementation an extra type of condition is introduced, this being a (destructive) overflow test. The symbol 'OVERFLOW' ('OVR') may be preceded by 'NO', in which case SETNO is executed setting ACCUMULATOR to 0, or 'NO' may be absent in which case SETYES is executed setting ACCUMULATOR to 1. The execution of TYPEPROC is irrelevant in this context.

Following the occurrence of the symbol 'OVERFLOW', the action OVRTEST is executed, which sets FUNCTION to #24. Thus it may be seen that tests are prepared as if-false-jumps, this being the test normally required following 'THEN', and are reversed as required (by REVCJ).

Comparisons

```
[Condition= . . . .
   (STACKEXPR,ANYPREF,Lcond,Rcond,RELATION)]

Lcond=(Axpr)
   (zero,CONSTANT)

Rcond=(ro,Q30)
   (SETNEZ)

Q30=(Condterm,Condterms)
   (zero,ZEROCOMP)

Condterm=(Term,CONDPLUS)
   (plus,Term,CONDPLUS)
   (minus,Term,CONDMINUS)

Condterms=()
   (plus,Term,CONDADD,Condterms)
   (minus,Term,CONDSUB,Condterms)
```

Instead of evaluating the expressions on either side of the comparator, and the performing the comparison by means of a subtraction, the two expressions are evaluated as one expression. In a simple case this is performed by effectively reversing the signs of all the terms of the right hand expression. This results in less instructions being generated, and usually one less accumulator being used. As the scale to which the expression is to be evaluated is not specified, it is determined by the compiler, using the normal method (see SCALETERM).

The expression to the left of the comparator is processed by the rule Lcond, and that on the right hand side by Q30. The first Term of this second expression causes either CONDPLUS or CONDMINUS to be executed, and subsequent Terms cause either CONDADD or CONDSUB to be executed. If however this second expression is a zero constant, the action ZEROCOMP is executed instead. This optimises cases of comparison against zero.

In this implementation, the comparator and second expression may be omitted. This allows the use of "pseudo" Boolean expressions, zero being false, and non-zero true. Thus the symbols <>0 are effectively assumed. This allows such conditions as :

```
'IF' 'BIT'[9]X 'THEN' . .
```

In this case the action SETNEZ is executed.

When the comparison is complete the action RELATION is executed. This sets up ACCUMULATOR and FUNCTION in preparation for a subsequent call of OUTCJ, and optimises tests of partwords against zero.

Statements

```
[Program=( . . ,Body, . . )

Body=(D1,ENDDECS,S1)]

S1=(St,Q39)

Q39=()
   (semi,S1)

St=(S,ENDST)

S=()
(id,Q38)
(STATUSCHECK,Q26,A1)
(Ifstat)
(Forstat)
(Gotostat)
(Compound)
(code,STATUSCODE,begin,Q35)
(Answerstat)

Q38=(SETLAB,colon,S)
   (LOOKUP,STATUSCHECK,Q33)
```

After processing each statement in the body of a program, the action ENDST is executed. This frees any store used temporarily during the statement. Statements are separated by semicolons (Q39), and may be void (S).

Where a statement commences with an identifier this may be a label, or the start of an assignment or procedure call. The first case is resolved in rule Q38. If the identifier is followed by a colon (:= having been converted to ←) the action SETLAB is executed before the colon is read. This action looks up the record of the label in the label list, and inserts the current address in this record, setting a marker to denote that this label is now set.

Assignment Statements

```
[S= . .
   (id,Q38)
   (STATUSCHECK,Q26,A1)
   . . . .

Q38= . . .
   (LOOKUP,STATUSCHECK,Q33)]

Q33=(Q25,A1)
   (LHSPROC,Proccall,BEHEAD)

Q25=()
   (Subscript)

Q26=(Bitset,Lhsb)
   (NONAME,Singlesubs)

Lhsb=(id,LOOKUP,LHSBITS,Q25)
   (NONAME,LHSBITS,Singlesubs)

A1=(becomes,VARCHECK,SETASS,A2,STORE,BEHEAD)
```

The rules for the left hand side of an assignment are complicated by the necessity to remove the cases of label settings and procedure calls, and the number of possibilities available.

The action STATUSCHECK is executed once, irrespective of which route is chosen, this completes any deferred actions associated with conditions and Answer statements, and checks that the statement can be entered.

An arithmetic operand record will have been set up, either by LOOPUP if an identifier has been processed, or by NONAME if an anonymous reference has been used. If the identifier was subscripted, or an anonymous reference has been used, the address fields of this record will have been processed by the actions associated with subscripts, and in a non-optimum case code may have been generated to evaluate the subscript expression(s).

If 'BITS' has been used the PARTWORD field of the record will have been set to the partword specification generated by Bitset, and after checking, its type will have been changed to integer, by LHSBITS. (Note that the assignment to a part of a partword table field is illegal.)

[A1=(becomes,VARCHECK,SETASS,A2,STORE,BEHEAD)]

A2=(A3)
(Expr)

A3=(Cexpr)
(string,STRINGEX)
(zero,STOREZERO)
(Term,ADDA,A4)
(plus,Term,ADDA,A4)
(minus,Term,SUBA,A4)

A4=(SIMPLEASS)
(SELOPTA,A5)

A5=(Q6a)
(Axpra)

After the left hand side of an assignment statement has been processed, the symbol becomes is read and the action VARCHECK executed. This ensures that the type of the left hand side is a data type, and that a subscript has been supplied where required.

The action SETASS is then executed. This sets up the variables required for an assignment, and also sets up the lowest expression level to the type of the left hand side. If 'ASSIGNMENT' 'TRACE' is required, the right hand expression must be evaluated in accumulator 7 and assignment optimisation inhibited. In this case SETASS returns a result of 1 to be used as a selector by A2.

If trace is not required, SETASS returns a result of zero, which causes rule A2 to select rule A3.

The assignment of zero is a special case which causes STOREZERO to be executed. Strings and conditional expressions are processed in the normal manner. The remaining cases are equivalent to those represented by the rule Axpra.

After the first Term has been processed its Arithmetic Operand record is compared with that of the left hand side, by ADDA or SUBA as appropriate, to detect whether an add to store type of assignment may be used. If no more Terms follow, the action SIMPLEASS is then executed to optimise certain cases of simple assignments. Otherwise the action SELOPTA is executed. This returns a zero result if an add to store assignment is not appropriate, and the rest of the expression is processed using rule Q6a. If however, an add to store assignment is to be used, a result of 1 is returned. This causes rule A5 to select the rule Axpra, which processes the rest of the expression, without using the previously processed first Term. (The difference between these two alternatives is vital to this optimisation.)

After the right hand side of the assignment has been processed the action STORE is executed. This outputs the instructions to perform the assignment, using the instruction number set up in ASSFUN, after generating trace instructions if required. The final action BEHEAD, removes the Arithmetic Operand record of the left hand side from the stack.

Conditional Statements

```
Ifstat=(if,IFS,Conditionlist,then,ENDST,St,else,ELSES,S,FI)
       {if,IFS,Conditionlist,then,ENDST,St,FI}
```

In this implementation, the syntax of a conditional statement differs from the Official Definition in that a conditional (or for) statement is allowed to follow 'THEN'. This avoids the necessity to use a 'BEGIN' - 'END' pair to surround the consequence statement in these cases. As processed by the syntax analyser generator, only the first alternative of the above rule is used. After checking the syntax, and generating the SYNTAX array, a one word overwrite introduces the second alternative.

```
(Overwrite #70450000 with #70450000+('SELF'+3-SYNTAX[0]) )
```

This would seem to allow:

```
'IF' . . 'THEN' 'IF' . . 'THEN' . . 'ELSE' . . ;
```

to be interpreted as either:

```
'IF' . . 'THEN' 'BEGIN' 'IF' . . 'THEN' . . 'ELSE' . . 'END' ;
```

or:

```
'IF' . . 'THEN' 'BEGIN' 'IF' . . 'THEN' . . 'END' 'ELSE' . . ;
```

This is the pathological "dangling else" from Algol 60. As implemented, the first interpretation is invariably chosen by the compiler. In cases of doubt, the user is advised to use 'BEGIN' - 'END' to remove apparent ambiguity.

After reading the symbol 'IF' the action IFS is executed. This prepares for the processing of the following condition, setting STATUS to 4 for jump optimisation. The action ENDST, which is executed after the symbol 'THEN' is read, serves to recover any workspace used temporarily during the evaluation of the conditions. The consequence statement is then processed. If this is followed by the symbol 'ELSE', the action ELSES is executed, and the alternative statement processed. At the end of the conditional statement, the action FI is executed. This ensures that any jump around the alternative statement will subsequently have its address set.

Goto Statements

```
Gotostat=(goto,id,Q27)

Q27=(GOTOL)
   (LOOKUP,STATUSCHECK,Singlesubs,LABSK,GOTOSK)
```

The destination of a Goto may be either a label or a switch entry. This is resolved by rule Q27.

If the destination is a label, the action GOTOL is executed. This looks up the label identifier in the local label list, and optimises such cases as 'THEN' 'GOTO', as the final test instruction of the condition has not yet been output.

The second case, that of a switch entry, is treated in a manner analagous to that of an expression. An Arithmetic Operand record is created by LOOKUP, and has subscript optimisation applied to it by the actions invoked by the processing of the subscript by Singlesubs. This record is then checked for type by LABSK, and finally the action GOTOSK is executed. This action outputs the indirect jump instruction, using the output procedure normally used for Arithmetic instructions, INST. This is not surprising, as the address used for this instruction will normally lie in the (special) data area of the segment.

For Statements

```
Forstat=(for,STARTFOR,Locvar,CHECKCV,becomes,F1)

F1=(A3,ASSCV,F2)

F2=(F3)
   (while,Conditionlist,WHILEL,F3)
   (step,Expr,STEPUNT,until,Expr,STEPUNT,F3)

F3=(do,DOCS,S,ENDFOR)
   (comma,MOREFOR,F1)
```

In regard to the structure of the code generated, for statements may be split into three categories, depending upon the complexity of the for-list. The simplest, and very common case, is where there is a single step-until element with three constants. In this case the initial value is first assigned to the control variable, the controlled statement executed, the control variable incremented, and finally its value tested against the limit value. A conditional jump back to repeat the controlled statement is made if the limit is not exceeded. This is the only case where the instructions for incrementation and testing of the control variable follow those of the controlled statement in core. In all other cases they precede those of the controlled statement, thus involving the requirement for a jump round the controlled statement when the for list is exhausted, and a jump back at the end of the controlled statement.

In the second case, the for list consists of one element of any type, or two elements where the first is simply an expression and the second a while element. In this case there is only one destination required for the jump back which is made at the end of the controlled statement.

In the last case, there is more than one destination required for the jump back at the end of the controlled statement. This statement is compiled as an anonymous procedure, which is called from the code generated for the for-list, and returns using the link supplied by the call. The detection and selection of these three cases is made using the variables FORSTATE (complexity) and CCC (detection of three constants).

After the symbol 'FOR' has been read the action STARTFOR is executed, which sets up and initialises a new for level. It also sets up a block level for labels, to ensure that the controlled statement cannot be illegally entered. The control variable is then processed by Locvar, which checks that it is correctly subscripted, if required. The control variable is further checked by CHECKCV which ensures that it is either a 'FIXED' or 'INTEGER' and is not a partword. This action also sets up the variables used by assignment optimisation, and expressions.

The first expression of each for element is processed by the rule A3, which optimises the assignment, where possible, which is output by the action ASSCV.

Where the element is of the form step-until, the two expressions are processed by the rule Expr, and after each, the action STEPUNT is executed. This ensures that these expressions are converted to the type of the control variable, and are stored in temporary locations if required.

Where the for element is of the type while, the following condition is processed by the rule Conditionlist, the final jump being output by WHILEL.

Where there is more than one for element, the action MOREFOR is executed following the comma. This releases any temporary locations used by step-until elements, and resets the variables for the following expression. This action also outputs the instructions required for the call of, or jump to, the controlled statement. In the case of a step-until element, this is preceded by a test of the control variable, and followed by its incrementation.

Following the symbol 'DO' the action DOCS is executed. In cases where the controlled statement is an anonymous procedure, this outputs the instruction to store the link. If 'LOOP' 'TRACE' is required, the instructions for the printing of the control variable are output at this point.

After the controlled statement has been processed the action ENDFOR is executed. This outputs a jump back as required, or the incrementation and testing of the control variable in the single element three constant case. All store used temporarily during the execution of the for statement is now released, the label block level reduced, and the for level reduced.

Blocks and Compound Statements

```
Compound=(STATUSCHECK,begin,Q37)

Q37=(S1,end)
   (BEGINBLOCK,Body,end,ENDBLOCK)
```

The distinction between blocks and compound statements is that, although they both start with 'BEGIN', this is only followed by declarations in the case of a block. This is resolved by rule Q37. There are no specific compiling actions associated with a compound statement, the 'BEGIN' and 'END' merely acting as "statement brackets".

In the case of a block however, the action BEGINBLOCK is executed after the symbol 'BEGIN' has been read. This increases the block level, and prepares for the following declarations. At the end of the block the action ENDBLOCK is executed. This reduces the block level, removes any records of identifiers declared within the block, and releases any data space allocated. Note however that the value of DATAMAX is not reset to its value prior to the block, if this value was smaller than its value at the end of the block.

Code Statements

```
[S= . . .
   (code,STATUSCODE,begin,Q35)
   . . . . . .]

Q35=(Csl,end)
   (BEGINBLOCK,Dl,ENDDECS,Csl,end,ENDBLOCK)

Csl=(Cs,Q34)

Q34=()
   (semi,Csl)

Cs=()
   (Compound,STATUSCODE)
   (inst,INSTTYPE,Npart,OUTCODE)
   (id,CODELAB,colon,Cs)
```

A code statement is bracketed by 'CODE' 'BEGIN' . . 'END'. In this implementation the 'BEGIN' may be followed by declarations, causing the code statement to be treated as a block, following the usual rules. This enables local workspace to be obtained without having to resort to the use of labelled locations, this being forbidden.

After any declarations, an item in a code statement can only be a code instruction, a compound statement, or a block. These may be labelled in the usual manner. It is not possible to include constants as code items. The 'SPECIAL' 'ARRAY' facility is provided to give an alternative method of including preset constants, possibly containing address information. The advantage of this is that not only is it more efficient, (no jump over being required) but it allows programs to be run with different O and N relativiser settings.

The Npart of a code instruction is not treated as an address but as an operand, in much the same manner as a primary. The actions which process the Npart leave on the stack an Arithmetic Operand record which is used by OUTCODE to output the instruction. This is provided for the convenience of the Coral programmer who wishes to use code (perhaps innocently by macro) rather than for the Astral programmer who wishes to use Coral.

```
[Cs= . . .
   (inst,INSTTYPE,Npart,OUTCODE)
     . . . . . .]

Npart=(Nread)
   (Nwrite)
   (Nshift)
   (Njump)
   (Nloc)

Nread=(Nwrite)
   (Sgint,CONSTANT)
   (CLEARTYPE,Type,orb,Pnumber,SCALECON,crb,CONSTANT)

Nwrite=(anyacc,NISACC)
   (id,LOOKUPD,Nw1)
   (NONAME,Nw2)

Nw1=()
   (Nw2)

Nw2=(osb,Nw3,INCTOS,csb)

Nw3=(Sgint)
   (modacc,NISMOD,Zint)

Zint=(Sgint)
   (ZERONUM)

Nshift=(int,CODESHIFT)
   (modacc,Zint,CODESHIFT,NISMOD)

Njump=(ID,LABL)
   (self,Zint,RELADD)

Nloc=(Nwrite)
   (string,STRINGEX)
```

The syntax of the allowable Npart of an instruction depends on the type of the instruction. The class to which an instruction belongs is returned as the result of INSTTYPE and is used as the selector for the rule Npart.

Where a constant is allowed as an operand, this is treated as a reference to a constant and not as a numeric address. The constants are optimised by the loader, which uses the functions 04-07 where possible. Where a constant is prefixed with a data type, the constant is converted to this type by SCALECON before being inserted in the Arithmetic Operand record set up by CONSTANT.

Where an identifier is used as the operand of a non-jump instruction, this is used by LOOKUPD in the usual manner. This may be followed by a modifier and/or a displacement, enclosed in square brackets, this construction also being allowed as the equivalent of an anonymous reference. Thus to obtain a numeric address, this must be enclosed in square brackets in exactly the same way in which it would have to be if it were used in a normal Coral expression.

The destination of a (direct) jump instruction can only be a label identifier, or a relative jump.

Table Declarations

```
Tabledec=(table,TYPEINT,TYPEARRAY,Newid,osb,int,TABLESIZE,csb,
                    Tabledetail,CLEARTYPE,TYPEINT)

Tabledetail=(osb,TABADD,Fieldlist)

Fieldlist=(CLEARTYPE,id,Fieldtype,TYPEARRAY,NEWNAME,Q60)

Q60=(csb)
   (semi,Q60a)

Q60a=(Q60)
   (Fieldlist)
```

After reading the symbol 'TABLE' the actions TYPEINT and TYPEARRAY are executed to set the type of the following table identifier, which is processed by Newid, to that of an integer array.

In the Official Definition both the number of words per entry, and the number of entries has to be specified. This implies that all subscripts must be multiplied by the width of the entry each time a field is referred to. This was felt to be an unjustifiable overhead for the Argus. Instead, only the total space required for the table is specified, and subscripts must be arranged to have the appropriate factor applied. This may often be done with no overhead. If for example, a for statement is used to scan the table, the step would be the entry width, instead of one. If entries are referred to by pointers, variable length and mixed record types may be kept in the same table. (As in the Compiler)

After the single bracketed integer specifying the size of the table has been read the action TABLESIZE is executed, this action reserving the required amount of data space. This is followed by the description of the fields, enclosed in square brackets. At the end of the declaration the actions CLEARTYPE and TYPEINT set IDTYPE to integer in case the table is to be preset.

```
Fieldtype=(Type,Sgint,FIELDDISP)
   (Partfield)
   (unsigned,Partfield,UNSFIELD)

Partfield=(orbosb,int,NOBITS,NOSIG,Intfixfield,crbcsb,
                    Sgint,FIELDDISP,commab,int,FIELDPOSN)

Intfixfield=(comma,Sgint,NOAFTER)
   (PARTINT)
```

A table field may occupy a whole word. In this case the field identifier is followed by a data type and a signed integer giving its word position in the entry. The action FIELDDISP calculates the address required for the field using the address TABLED and TABLEI stored by TABADD, and inserts it into the appropriate locations at the base of the stack in readiness for the subsequent call of NEWNAME.

A table field which occupies part of a word may be stored either with a sign bit, or without, in which case 'UNSIGNED' is specified. The type of the field is either integer or fixed, this being resolved by rule Intfixfield. It should be noted, particularly in the case of an integer field, that the number of significant bits is taken to be the number of bits occupied, plus one in the case of an unsigned field. The action FIELDPOSN is executed after the integer specifying the starting bit position has been read, this completing the part word specification. It should be noted that Ferranti bit numbering is used, and the starting bit position specifies the most significant bit of the field. At the request of Ferranti, either square or round brackets may be used in the specification of the field type.

Switch Declarations

```
[D= . . . . . .
   (switch,ADDRSW,TYPESWITCH,Newid,becomes,Swlist)
    . . . . . .]

Swlist=(id,SPECLAB,Q41)

Q41=()
   (comma,Swlist)
```

A switch declaration sets up, in the segment's Special Data area, a set of sequential locations containing the addresses of the labels, one per entry. Where a label is external to a segment, the address will be that of the indirect jump to the label, set up at the end of the segment. The action ADDRSW sets the direct address of the record being formed, to the current special data transfer address minus one. Its type is set by TYPESWITCH, and is effectively Label Array.

After each label identifier in the switch list, the action SPECLAB is executed. This looks up the label in the (local) label list, and outputs either zero or its chain address to special data. (The label cannot yet have been set at this block level, except in the special case of label parameters.)

Overlay Declarations

```
Overlaydec=(overlay,Base,with,OVERON,Datadec,OVEROFF)

Base=(id,LOOKUPD,Q43)
   (NONAME,osb,int,INCTOS,csb)

Q43=()
   (osb,Sgint,Inctos,csb)

Datadec=(Type,Q21)
   (Tabledec)
```

Any data location may be overlayed with a data declaration. The base defines the starting point of the overlaying data. As an extension, this may be an (absolute) anonymous reference, thus giving a limited local equivalent of an 'ABSOLUTE' communicator.

An arithmetic Operand record is created by either NONAME or LOOKUPD. In the second case, if the direct address is zero, the indirect address replaces it. The action INCTOS increments the direct address by the integer constant.

The data declaration is preceded by the execution of the action OVERON, which deletes the Arithmetic Operand record stores its direct address in OVERBASE, and sets the OVERLAY flag. This flag is cleared later by OVEROFF.

The following points should be noted:

1 Where a multi-dimensional array is overlayed, the actual area overlayed is its first level Iliffe vector. It is usually better to overlay a single dimensional array with multi-dimensional array(s).

2 There is no check to ensure that the space required by the overlaying data declaration does not exceed that of the overlayed data area.

3 Where a procedure parameter is overlayed, the base is taken as the location occupied by the parameter within the data space local to the procedure, even if the parameter is a 'LOCATION' or 'ARRAY' pointer.

4 Where the base is a field of a table passed as a parameter, the effective address is the displacement of the field relative to the start of the table.

Special Declarations

```
Specialdec=(special,array,TYPEINT,TYPEARRAY,ADDRSPEC,Newid,becomes,Speclist)

Speclist=(Specitem,Q46)

Q46=()
   (comma,Speclist)
```

The Special Array facility is introduced as an aid to the production of software and not as a facility for everyday use. Its introduction compensates for the restriction that constants may not be embedded within code. A special array is classed as an Integer Array, with a zero lower bound, and must be completely preset. The normal restrictions on presetting do not apply. The preset values may contain address and numeric information packed into one word (10/14 bit packing). It is usually bad practice to alter the contents of a special array during the execution of a program.

```
Specitem=(Zint,Q45)
   (string,SPECSTRING)
   (CLEARTYPE,Type,orb,Pnumber,SPECNUM,Morespec)

Morespec=(crb)
   (comma,Pnumber,SPECNUM,Morespec)

Q45=(SETTEN,Specadd)
   (CLEARTYPE,TYPEINT,SPECNUM)

Specadd=(colon,id,SPECLAB)
   (self,Sgint,SPECREL)
   (div,Q44)

Q44=(Sgint,NONAME,INCTOS,SPECCON)
   (Base,SPECCON)
```

A special item may be either a packed word, a string, or a numeric constant. If the constant is not to be stored as an integer, one or more constants, enclosed in round brackets may be preceded by a data type. These constants will be rescaled by SPECNUM before being output. In the case of a string, the actual standard string, occupying one or more words, is stored in the special array at the point of its occurrence by SPECSTRING. This differs from presetting an integer with a string, where the preset value is its address.

Where a packed word occurs, the action SETTEN is executed to store the ten bit constant temporarily in TOPTEN. This constant, which may be void, is then followed by either a colon, the symbol 'SELF' (*), or an oblique stroke. The colon must be followed by an identifier, which is taken as that of a label by the action SPECLAB, and the label address subsequently stored.

In the case of a relative address ('SELF' or *), the symbol is followed by a signed integer, the action SPECREL treating the address as relative to the current special data transfer address. (NB not the current program transfer address.) In the third case, the fourteen bit field may be either a second integer constant, or a data address, as processed by the rule Base. The Arithmetic operand record set up in this case is used by the action SPECCON.

Level Directives

```
[Compileitem= . . . .
   (Leveldec)
   . . . . . .

D= . . . . .
   (Leveldec)]

Leveldec=(level,int,SETLEVEL)
```

A level directive sets the level of diagnostic information required at compile time, and required to be output to the loader. After the unsigned integer giving the level required, the action SETLEVEL is executed.

Trace Directives

```
[Compileitem= . . . .
   (Yesno,Tracetype,trace)
   . . . . . .

D= . . . .
   (SETYES,TYPEPROC,Q48)
   (no,SETNO,Tracetype,trace)
   . . . . . .]

Tracetype=(assignment,ASSTRACE)
   (loop,FORTRACE)
   (label,LABTRACE)
   (procedure,PROCTRACE)

Q48=(assignment,ASSTRACE,trace)
   (loop,FORTRACE,trace)
   (label,LABTRACE,trace)
   (procedure,Q49)

Q49=(Newid,BEGINPROC,Procrest)
   (PROCTRACE,trace)
```

Trace directives govern the amount of instructions specifically generated to enable the action of a program to be traced. Trace is split into four categories; assignment, loop, label and procedure. Each type may be turned on or off as required, the trace state following the block level. At the end of a block the state is reset to that of the outer block, but at the start of a block it is initially that of the outer block. Normally it is off.

The actions SETYES and SETNO set the value of ACCUMULATOR to 1 and 0, respectively. (These actions also being used for overflow tests.) The actions ASSTRACE, FORTRACE, LABTRACE and PROCTRACE setting the appropriate bit of TRACE from the value of ACCUMULATOR.

Because of the clash (starting with same symbol) between 'PROCEDURE' 'TRACE' directives and the declaration of untyped procedures, the rules have been expanded by SID, and the action TYPEPROC is always executed in conjunction with the action SETYES. A manual transformation of the above rules could reduce their size.

Trace instructions are generated by the actions STOREAWAY, DOCS, SETLABEL, and PROCENTRY & EXITCHECK.

Procedure Type Declarations

```
[D=(Type,Q47)
   (Valproc,Newid,BEGINPROC,colon,Newid,NEXTPSET,Procrest)
   (SETYES,TYPEPROC,Q48)
   . . . . . .

Q47= . . .
   (procedure,Q49)

Q49=(Newid,BEGINPROC,Porcrest)
   . . . . . .]

Valproc=(value,TYPEVPROC,procedure)
```

The above rules "untangle" the start of a procedure declaration. Before the procedure identifier is read, the type information in IDTYPE is complete, so that NEWNAME can move the Identifier Specification Record onto the stack. At this point however, the PARAMSPEC pointer has not been set, and the DIRADD field contains the address of the location which will subsequently be used to hold the link. DIRADD[0] is set correctly however for allocating a location for the first parameter.

After the procedure identifier has been read, the action BEGINPROC is executed, this completing the procedure's Identifier Specification record, and initiating its Parameter Specification record. A location in special data is allocated as a pointer to the procedure, and the address of this inserted into the DIRADD field of its record. In preparation for the following parameters, if any, the PAC field of IDTYPE is set to 7.

If the procedure is a 'VALUE' procedure, its name is followed by a colon, and the identifier of the "context" parameter, which is processed by Newid. This is followed by the execution of NEXTPSET to clear the type and special markers from IDTYPE, leaving the PAC field set to 6.

Procedure Parameter Declarations

```
Procrest=(Parameterpart,semi,Procbody,ENDPROC)

Parameterpart=()
   (orb,Parameters,crb)

Parameters=(Parameterset,Q65)

Q65=()
   (semi,NEXTPSET,Parameters)

Parameterset=(Type,Q63)
   (location,TYPELOC,Q62)
   (label,TYPELAB,Idlist)
   (switch,TYPEISWITCH,Idlist)
   (table,TYPEINT,TYPEIARRAY,Newid,osb,int,csb,PARAMTAB,Tabledetail,PARAMTAB)

   (value,Q64)
   (SETYES,TYPEPROC,procedure,Procparamlist)

Q64=(Type,Idlist)
   (TYPESPEC,Pairlist)
   (TYPEVPROC,procedure,Procparamlist)

Pairlist=(Newid,colon,Newid,Q61)

Q61=()
   (comma,Pairlist)

Q62=(Type,Idlist)
   (TYPESPEC,Pairlist)

Q63=(array,TYPEIARRAY,Idlist)
   (TYPETPROC,procedure,Procparamlist)

Procparamlist=(Newid,Paramlev2,Q58)

Q58=()
   (comma,Procparamlist)
```

The type of a set of parameters is built up incrementally in IDTYPE, the PAC field having been previously set. In particular, where the location of an object is to be passed as a parameter the marker LCM will be set. Each parameter identifier will cause the action NEWNAME to be executed. As well as adding the Identifier Specification record to the DECLIST, in the case of parameters NEWNAME will call NEXTPARAM to add the specification of the parameter to the procedure's Parameter Specification record.

When the procedure has been compiled, the Identifier Specification records of the individual parameters will be deleted, as they are then out of scope. The Parameter Specification record of the procedure will however remain until the procedure itself goes out of scope. This record, and not the records of the individual parameters, is required for the generation of calls of the procedure.

In the cases of typed 'VALUE' & 'LOCATION', 'ARRAY', 'SWITCH' and 'LABEL' parameters the specification is followed by a list of parameter identifiers, separated by commas. In the case of untyped 'VALUE' & 'LOCATION' parameters, the identifiers occur in pairs, the second of which is used to pass the type/scale of the first. 'TABLE' parameters require the size of the table to be specified, even though this is ignored. This is followed by the specification of the individual fields.

In the case of typed or untyped 'PROCEDURE' parameters, the parameter identifier is followed by a specification of its parameter requirements, if any. This enables a Parameter Specification record to be formed for each 'PROCEDURE' parameter, this being required when a call of the parametric procedure is to be compiled.

After the semicolon separating sets of parameters has been read, the action NEXTPSET is executed, clearing the type information from IDTYPE in readiness for the next set.

Procedure Parameter Specifications

```
Paramlev2=(BEGINPSPEC,Q57)

Q57=(orb,Paramtypelist,crb,ENDPSPEC)
   (ENDPSPEC)

Paramtypelist=(Paramtype,NEXTPARAM,Q56)

Q56=()
   (comma,NEXTPSET,Paramtypelist)

Paramtype=(Type,Q54)
   (location,TYPELOC,Q53)
   (label,TYPELAB)
   (switch,TYPEISWITCH)
   (table,TYPEINT,TYPEIARRAY)
   (value,Q55)
   (SETYES,TYPEPROC,procedure)

Q55=(Type)
   (TYPESPEC,NEXTPARAM)
   (TYPEVPROC,procedure)

Q53=(Type)
   (TYPESPEC,NEXTPARAM)

Q54=(array,TYPEIARRAY)
   (TYPETPROC,procedure)
```

Where a procedure is specified without being declared, this occurring in the case of 'PROCEDURE' parameters in a procedure declaration, and also where procedures are specified in communicators, its parameter requirements must also be specified if the procedure requires parameters. This enables a Parameter Specification record to be prepared, and a reference to it inserted in the PARAMSPEC field of the Identifier Specification record of the procedure. This record is required even if the procedure (apparently) has no parameters. (NB A "parameterless" 'VALUE' procedure has the "context" parameter.)

In the rule Paramlev2 the action BEGINSPEC is executed to initiate the Parameter Specification record, tentatively reserving eight words for this. The rule Q57 detecting the case where the procedure requires no parameters. In this case the action ENDPSPEC being immediately executed. These two actions setting up a null record.

Where parameters are required, an opening round bracket will be read. This is followed by a list of parameter types, separated by commas, and terminated by a closing round bracket; after which the action ENDPSPEC is executed.

The type of each parameter is built up incrementally in IDTYPE, and added to the record by the action NEXTPARAM. After each comma the type information is cleared from IDTYPE by NEXTPSET. In the case of untyped 'VALUE' & 'LOCATION' parameters the action NEXTPARAM is effectively executed twice, to insert the specification of the context parameter. Where procedures are specified at this level, a specification of their parameters is not required.

The Procedure Body

```
Procbody=(code,begin,KILLPARAMS,Od1,ENTERPROC,Cs1,end)
   (SETPARAMS,Q51)

Q51=(begin,Od1,PROCENTRY,S1,end,EXITCHECK)
   (ENDDECS,PROCENTRY,Ss,EXITCHECK)

Od1=(D1,ENDDECS)
   (ENDDECS)

Ss=()
   (Ifstat)
   (Forstat)
   (Answerstat)
   (Gotostat)
   (Id,LOOKUP,STATUSCHECK,Q33)
   (STATUSCHECK,Q26,A1)
```

Where the body of a procedure is a code statement, the normal parameter machanism is inhibited by the action KILLPARAMS. This cancels the allocation of data space for the parameters, and deletes their Identifier Records.

Otherwise the action SETPARAMS is executed, which confirms the allocation of the data space required to hold the parameters. If the procedure body is a block, a new block level is not set up, as one has already been set up for the procedure. This prevents the redeclaration of the identifiers used for parameters, at the outer level, thus rendering them inaccessable.

When the first statement of the procedure body is reached, the action PROCENTRY is executed. This outputs instructions to store the link and the parameters. At the end of the procedure the action EXITCHECK is executed. This ensures that a result is returned if a result is required.

Answer Statements

Answerstat=(STATUSCHECK,answer,SETANS,Expr,ANSCHECK)

This statement is only allowed within the body of a typed procedure. After the symbol 'ANSWER' has been read, the action SETANS is executed. This sets EXPSCALE to the type required for the result, and sets SCALEFIRM. If however the procedure is a 'VALUE' procedure, these are both cleared.

At the end of the expression delivering the result, the action ANSCHECK is executed. This ensures that the result has been evaluated in accumulator 7, to the scale given in EXPSCALE. The variable STATUS is set to 2 to denote that an exit instruction is required.

Procedure Calls

```
[Tpri= . . . . . .
   (id,LOOKUP,Q28)
   . . . . . .

Q28= . . . . . .
   (RHSPTEST,Rhspsel)

Rhspsel= . . . . .
   (Proccall)

S= . . . . . .
   (id,Q38)
   . . . . . .

Q38= . . . . .
   (LOOKUP,STATUSCHECK,Q33)

Q33= . . . . .
   (LHSPROC,Proccall,BEHEAD)]

Proccall=(SETUPPROC,Selparams,CALLPROC,FINISHPROC)

Selparams=()
   (orb,Paramloop)

Paramloop=(NEXTPTYPE,Selptype,ANYMORE,Selmore)

Selptype=(Expr)
   (Locvar)
   (id,LOOKUP)
   (id,Q32)

Selmore=(Multitest)
   (comma,Paramloop)

Multitest=(crb)
   (comma,CALLPROC,MULTICALL,Paramloop)
```

When the requirement for a procedure call is detected, the rule Proccall is entered and the action SETUPPROC executed. This prepares for the call of the procedure, and raises the expression level in readiness for the evaluation of parameters. If the procedure requires parameters a result of one is returned, otherwise zero. This value is used as the selector for the rule Selparams.

After the call of the procedure has been generated the action FINISHPROC reduces the expression level, and alters the type of the Arithmetic Operand record set up for the procedure identifier, to the type of the result of the procedure.

Before each parameter is processed, the action NEXTPTYPE is executed. This prepares for the evaluation of the parameter to the required scale, in the required accumulator. It returns a result, given by PARAMCLASS, which is used as a selector by the rule Selptype. This selects the syntax for the parameter, choosing the rule containing the actions appropriate to the type required.

After each parameter has been processed, the action ANYMORE is executed. This checks that the parameter is of exactly the required type. It returns a result of zero if no more parameters are required, and a result of one otherwise. This is used as a selector by the rule Selmore.

If more parameters are supplied than are required, this case being detected by the rule Multitest, it is assumed that a multiple call of the procedure is required. The call is output by the action CALLPROC, which in this case is followed by the action MULTICALL, which prepares for the second and subsequent calls.

Common Communicators

```
[Compileitem= . . . .
   (Commondec)
   . . . . . .]

Commondec=(common,orb,COMON,Commonlist,COMOFF,crb)

Commonlist=(STARTDEC,Commonitem,Q23)

Q23=()
   (semi,Commonlist)
```

A common communicator is compiled to produce a loadable segment. This consists of two main parts, a common data part providing data space accessible to all the segments, and a reference part containing references to switches, labels, and procedures accessible to all the segments, each being provided by only one segment.

In this implementation, a means of checking that all these references have been set before the programs are entered is provided by the "common check tape". Within the common segment all internal references are made by means of D and S address tags, but all references by segments to the common segment are made by means of C tags.

A checksum is formed from the characters of the common communicator to ensure that a segment is loaded with the correct common.

Common Data

```
Commonitem=()
   (Type,Q22)
   (Tabledec,Presetlist)
   (Overlaydec)
   (Specialdec)
   . . . . . .

Q22=(Q21,Presetlist)
   . . . . . .

[Q21=(Idlist,DECSIZE)
   (array,TYPEARRAY,Arraylist)]
```

Data declarations within a common communicator are handled in exactly the same manner as data declarations in a program segment. Where multi-dimensional arrays are used, the Iliffe vectors are set up in the special data area in the usual way. This also applies to strings and Specialdecs. The identifiers of these items of data are originally addressed relative to the D and S areas of the common area, these addresses being subsequently changed to addresses relative to the C area. It is important to note that this re-addressing assumes that the S area of the C area immediately follows the D area.

Common References

```
Commonitem=()
   (Type,Q22)
   . . . . . .
   . . . . . .
   (Valproc,Comprocs)
   (label,TYPELAB,Comspecidlist)
   (switch,TYPEISWITCH,Comspecidlist)
   (SETYES,TYPEPROC,procedure,Comprocs)

Q22= . . . . .
   (TYPETPROC,procedure,Comprocs)

Comspecidlist=(Comspecid,Comspecidcont)

Comspecidcont=()
   (comma,Comspecidlist)

Comspecid=(ADDRSPEC,Newid,SPECONE)

Comprocs=(Comspecid,Paramlev2,Q20)

Q20=()
   (comma,Comprocs)
```

Each identifier referring to an item to be provided by a segment, is processed by the rule Comspecid. This addresses the identifier with the current value of the special data transfer address, and after the identifier has been processed by Newname, outputs a preset value of zero to the allocated location by the execution of the action SPECONE.

Where an identifier is used in a label context in a special array in a common communicator, this reference is chained up to the previous specification or reference if one exists. In the case where one does not, an implicit specification of the identifier as a common label is made.

Absolute Communicators

```
[Compileitem= . . . .
   (absolute,orb,Abslist,crb)]

Abslist=(CLEARTYPE,Absitem,Q11)

Q11=()
   (semi,Abslist)

Absitem=()
   (Type,Q10)
   (Valproc,Absprocs)
   (label,TYPELAB,Absidlist)
   (switch,TYPESWITCH,Absidlist)
   (table,TYPEINT,TYPARRAY,Absid,osb,int,csb,Tabledetail)
   (SETYES,TYPEPROC,procedure,Absprocs)

Q10=(Q9,Absidlist)
   (TYPETPROC,Absprocs)

Q9=()
   (array,TYPEARRAY)

Absidlist=(Absid,Q7)

Q7=()
   (comma,Absidlist)

Absid=(id,div,int,ABSADD,NEWNAME)

Absprocs=(Absid,Paramlev2,Q8)

Q8=()
   (comma,Absprocs)
```

The purpose of the absolute communicator is to communicate to the compiler the specifications and addresses of items which may be regarded as having fixed core store locations. Each identifier is followed by its absolute address, this being processed by the rule Absid. This communicator may be used to enable peripherals to be referred to by an appropriate identifier.

The interpretation of the absolute address in the case of single word variables, and single dimensional arrays and tables is straightforward. In the case of multi-dimensional arrays, the absolute address is taken to be that of the zeroth element of the first level Iliffe vector. In the case of switches the address is taken to be that of the zeroth, not first, entry in the switchlist, and should therefore be set one back if the switchlist is regarded as having a lower bound of one.

In the cases of labels and procedures the absolute location specified is regarded as containing a pointer to the appropriate point in the program.

Library Communicators

```
[Compileitem=(library,Q69)]

Q69=(orb,Liblist,crb)
   (CLEARTYPE,Q66)

Liblist=(CLEARTYPE,Libitem,Q19)

Q19=()
   (semi,Liblist)

Libitem=()
   (Type,Q18)
   (Valproc,Libprocs)
   (SETYES,TYPEPROC,procedure,Libprocs)

Q18=(Libidlist)
   (TYPETPROC,procedure,Libprocs)

Libidlist=(Libid,Q16)

Q16=()
   (comma,Libidlist)

Libid=(id,div,int,LIBADD,NEWNAME)

Libprocs=(Libid,Paramlev2,Q17)

Q17=()
   (comma,Libprocs)
```

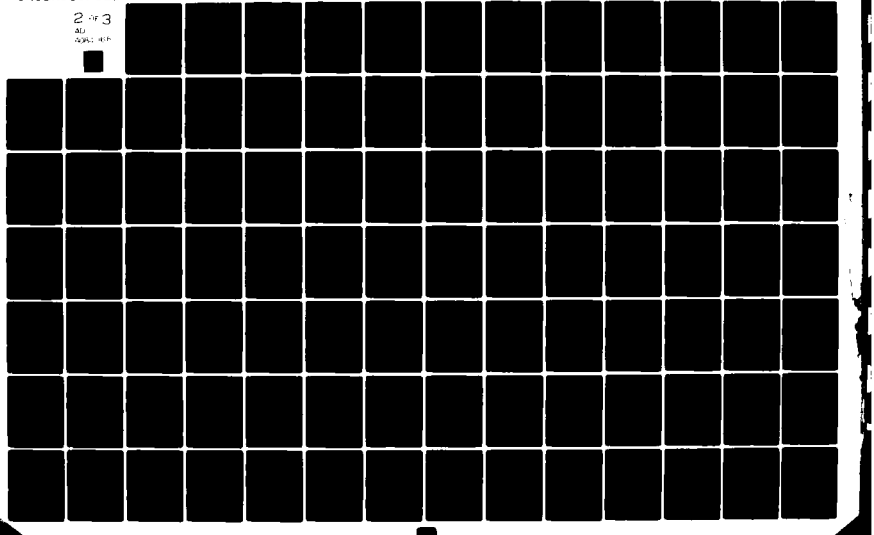
The purpose of a library communicator is to communicate to the compiler the names, specifications, and reference numbers of procedures and preset variables available in the library. Thus library procedures explicitly referred to do not have to be built-in to the compiler, necessitating its alteration each time the library is extended. Private libraries may be built up as required without difficulty. Treating library procedures in exactly the same manner as any other procedure avoids the necessity for placing restrictions on the use of library procedures as parameters of procedure calls. The special library procedures for tracing and floating point arithmetic are never explicitly referred to by the user, and do not need specification in a communicator.

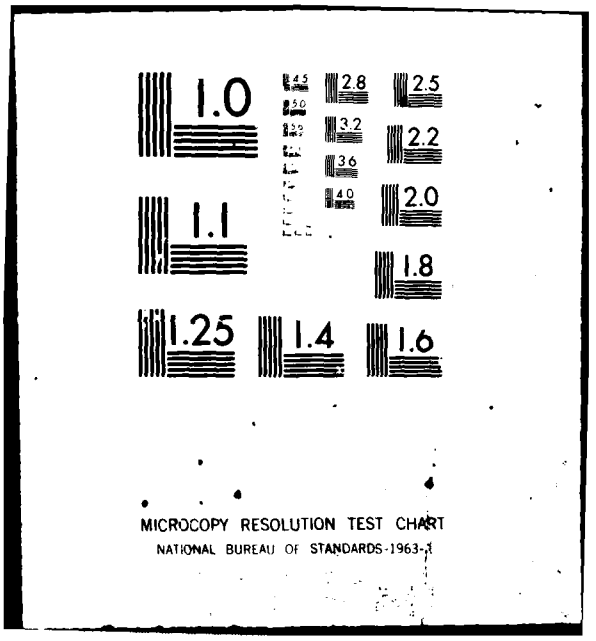
Library Compilation

```
[Compileitem=(Library,Q69)

Q69= . . . . .
   (CLEARTYPE,Q66)

Libid=(id,dir,int,LIBADD,NEWNAME)]

Q66=(Type,Q67)
   (Valproc,Libid,SETLIBSEG,colon,Newid,NEXTPSET,Procrest,FINISHSEG)
   (SETYES,TYPEPROC,procedure,Libid,SETLIBSEG,Procrest,FINISHSEG)

Q67=(Libid,becomes,Pnumber,SCALECON,SETLIBVAR)
   (TYPETPROC,procedure,Libid,SETLIBSEG,Procrest,FINISHSEG)
```

AD-A084 068 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
THE CORAL 66 COMPILER FOR FERRANTI ARGUS 500 COMPUTER.(U)
JUN 78 B GORMAN
UNCLASSIFIED RSRE-TN-799 DRIC-BR-67199 NL

2 OF 3
AD
A084 068




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A library procedure referring only to previously compiled procedures and variables, may be compiled when required. The RLB output tape in this case should be spliced on to the front of the library tape. After the procedure or variable has been compiled its specification remains set up for the duration of the run of the compiler. For subsequent runs however, its specification should be added to the library communicator.

Each new item added to the library is required to be allocated a reference number in the range 0 to 16383. This is the number by which it will be referred to by the loader, and is required by the rule Libid to follow the identifier of the library item in communicators and when compiling library items. The user however refers to the library item always by its identifier only.

The only two compiling actions specific to library compilation are SETLIBSEG for procedures, and SETLIBVAR for variables. The action LIBADD is used in both library compilation and library communicators, and is the only action specific to the second case.

External Communicators

```
[Compileitem= . . . .
   (extenral,orb,Extlist,crb)]

Extlist=(CLEARTYPE,Extitem,Q15)

Q15=()
   (semi,Extlist)

Extitem=()
   (Type,Q14)
   (Valproc,Extprocs)
   (label,TYPELAB,Extidlist)
   (switch,TYPESWITCH,Extidlist)
   (table,TYPEINT,TYPEARRAY,Extid,osb,int,csb,Tabledetail)
   (SETYES,TYPEPROC,procedure,Extprocs)

Q14=(Q9,Extidlist)
   (TYPETPROC,procedure,Extprocs)

Extidlist=(Extid,Q12)

Q12=()
   (comma,Extidlist)

Extid=(id,div,SETTEN,int,Zint,EXTADD,NEWNAME)

Extprocs=(Extid,Paramlev2,Q13)

Q13=()
   (comma,Extprocs)
```

An external communicator is similar in function and form to an absolute communicator. The difference being that whilst the address of an item must be specified before compilation in the case of an absolute communicator, it need only be specified before load time in the case of an external communicator.

This is performed by allocating 32 "external bases" (numbered 0-31) and allowing an item to be addressed up to $\pm$255 from the specified base. This is specified in the rule Extid.

The first integer specifies the base, and the second optional integer the displacement. Care should be taken that a reference to an item such as an external array does not exceed the displacement as there will be overflow into the field used for the base number. If this case occurs, it may be necessary to set sufficient adjacent bases at intervals of 512.

THE SYNTAX ANALYSER

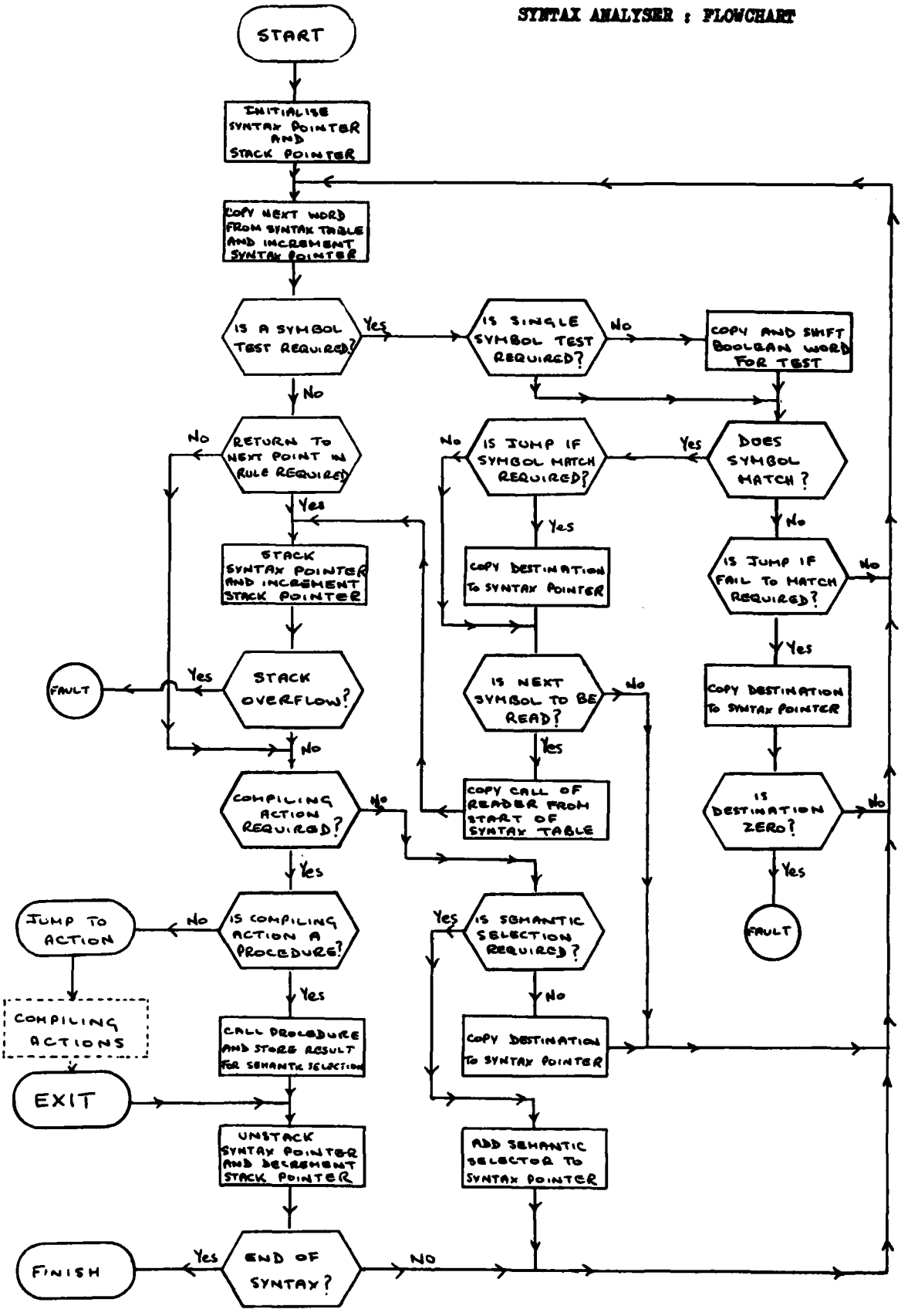
The syntax analyser is the main controlling routine of the compiler. By interpreting the processed syntax stored in SYNTAX, together with the auxiliary information stored in BOOLWORD, it not only checks the incoming source program for validity but invokes the required compiling actions at the requisite points.

These two tables have been produced automatically by the use of the SID and SAG syntax processing programs. These both check that there is no possible ambiguity in the definition of the syntax. The original syntax rules have been reduced to "one track form" by the use of SID, this obviates the requirement for backtracking, which can be time consuming. This produces a legible transformed syntax, which is used as the actual working syntax by the compiler writer, in this case, and has been edited as the compiler developed. The working syntax is checked and transformed into the two 'SPECIAL' 'ARRAY'S by the use of an Argus version of SAG (#BSAG) which runs on an ICL 1900.

The syntax table produced can be regarded as the machine code of a special purpose machine. In this interpretation, the syntax analyser is the machine, the syntax its "high level" language, SAG its compiler, and the Coral compiler its assembler. The importance of this approach lies in the fact that the syntax rules input to SAG are not merely used as the bare bones of the language definition, but are used as a high level language to express the overall compiling strategy.

A flowchart is given for the syntax analyser. This expresses unambiguously what its operations are, but should be used in conjunction with the following definition of the interpretive code in order to gain a fuller understanding of its operation.

SYNTAX ANALYSER : FLOWCHART



The interpretive instructions fall into two groups, symbol tests (m.s. bit=1), and unconditional operations (m.s. bit=0)

Unconditional Operations

In these operations the most significant two octal digits specify the operation, and the least significant five octal digits the "address". This may either be an address relative to the start of the syntax (s), or a core location (n).

Operation Code	Action
00	Goto syntax at s
04	Select one of the following (s) alternatives depending on the value stored in SSEL
10	Goto the labelled action n, taking the stacked interpretive link on return
14	Call the procedure whose address is stored at n, storing its result in SSEL and taking the stacked interpretive link
20	Call the rule s, stacking the interpretive link for return
30	Goto the labelled action n, continuing with the following interpretive instruction on return
34	Call the procedure whose address is stored at n, storing its result in SSEL and continuing with the next interpretive instruction

The actual representation of these operations in the output of SAG is as follows, where dd represents a decimal integer and id an identifier.

Operation Code	Representation
00	#000/dd
04	#100/dd
10	#200:id
14	#300/id
20	#400/dd
30	#600:id
34	#700/id

There are three special cases generated:

1 #600:READER to call the reader, at the start of the syntax to initialise T1, and subsequently to read further symbols.

2 #200:EXIT to exit from the end of an alternative of a rule. (Uses stacked link)

3 #200:FAIL to indicate syntax failure.

CONDITIONAL OPERATIONS

In these operations the most significant octal digit specifies the operation, the next three digits a terminal symbol or symbol group (b), and the least significant four octal digits an address relative to the start of the syntax (s). A special case occurs where the s field is zero, a successful jump in this case indicating the discovery of a syntactic error in the source program.

Where b is less than 72 a symbol match is defined to occur if the value of b equals that of the terminal symbol stored in T1. Where b is 72 or greater (in steps of three) it refers to a three word group in BOOLWORD. A symbol match is defined to occur if the T1 th bit of this group is set.

As a side effect of a conditional operation, a terminal symbol may be "accepted". This causes the next symbol to be read, by means of a jump to READER. This is not done directly, but by interpreting a type 30 instruction stored at the start of the syntax. This is done in order to reduce the size of the syntax analyser, although it is slightly slower.

Operation Code	Action
4	If symbol match jump to s.
5	If symbol match read next symbol and jump to s.
6	If no symbol match jump to s
7	If symbol match read next symbol, otherwise jump to s.

All operations of this type are represented by eight digit octal numbers. (#cbbbssss)

The Working Syntax Introduction

The syntax which follows is the actual syntax used by the compiler. It has been transformed from the original syntax by SID. This has introduced the auxiliary rules Q--. The following convention has been used for the identifiers, to aid the reader:

1	Rule Names	First letter in upper case, subsequent letters in lower case. The rule name is underlined where semantic selection is carried out.
2	Terminal Symbols	Lower case letters
3	Compiling Actions	Upper case letters. The action name is underlined where it delivers a result to be used as a semantic selector.

In the notation used for terminal symbols, lower case letters are used. Most of the symbol names are self explanatory, but a list of the more cryptic and composite symbols is given below.

zero	zero constant (of any form)
int	integer constant, including zero
intcon	integer constant, excluding zero
realcon	real constant, excluding zero and integer constants
pconst	zero, integer, or real constant
shift	shift operator (eg 'SLA')
ro	comparator (eg =, 'EQ')
inst	accumulator and function of code (eg 7 ADD)
colon	:
semi	;
orb	(
orbosb	(, [
crb	)
crbcsb	) ,]
mult	*
self	'SELF' , *
div	/
step	'STEP' , :
until	'UNTIL' , :
osb	[
csb	]
power	↑ , **
becomes	← , :=
commab	'BIT' or ,
id	identifier

Syntax 1 Segments

```
Run=(Compileitem,Q70)

Q70=(finish)
   (semi,Run)

Compileitem=()
   (Program)
   (Commondec)
   (library,Q69)
   (external,orb,Extlist,crb)
   (absolute,orb,Abslist,crb)
   (Leveldec)
   (Yesno,Tracetype,trace)
   (test,SETTEST,Q68)

Q68=(Program)
   (Commondec)
   (library,CLEARTYPE,Q66)

Program=(id,BEGINPROG,begin,Body,ENDPROG,end)

Body=(D1,ENDDECS,S1)

D1=(STARTDEC,D,semi,Q50)

Q50=()
   (D1)
```

Syntax 2 Declarations

```
D=(Type,Q47)
   (Tabledec,Presetlist)
   (switch,ADDRSW,TYPESWITCH,Newid,becomes,Swlist)
   (Specialdec)
   (Overlaydec)
   (Valproc,Newid,BEGINPROC,colon,Newid,NEXTPSET,Procrest)
   (SETYES,TYPEPROC,Q48)
   (no,SETNO,Tracetype,trace)
   (Leveldec)

Type=(integer,Q59)
   (fixed,orbosb,int,NOBITS,comma,Sgint,NOAFTER,crbcsb)
   (floating,TYPEFLOAT)

Q59=(osb,int,NOBITS,csb,PARTINT)
   (TYPEINT)

Sgint=(int)
   (plus,int)
   (minus,int,NEGNUM)

Q47=(Q21,Presetlist)
   (TYPETPROC,procedure,Newid,BEGINPROC,Procest)

Q21=(Idlist,DECSIZE)
   (array,TYPEARRAY,Arraylist)

Idlist=(Newid,Q52)

Q52=()
   (comma,Idlist)

Newid=(id,NEWNAME)

Presetlist=()
   (becomes,SKIPDTA,Presets)

Presets=(Presetnum,Q40)

Q40=()
   (comma,Presets)

Presetnum=(Pnumber,OUTPRESET)
   (orb,Presets,crb)
   (string,PRESETSTRING)

Pnumber=(pconst)
   (plus,pconst)
   (minus,pconst,NEGNUM)
   (ZERONUM)

Arraylist=(ZEROARRAYS,Idlist,osb,Boundpair,Morebounds,csb,ENDARRAY,Q42)

Q42=()
   (comma,Arraylist)

Boundpair=(Sgint,SETLB,colon,Sgint,SETUB)

Morebounds=(ONEDIM)
   (FIRSTDIM,comma,Boundpair,Arraytail)

Arraytail=(LASTDIM)
   (MIDDIM,comma,Boundpair,Arraytail)
```

Syntax 3 Expressions

```
Expr=(Cexpr)(Axpr)
   (zero,CONSTANT,SCALETERM)
   (string,STRINGEX,SCALETERM)

Axpr=(Term,Q6)
   (Axpra)

Axpra=(plus,Term,Q6)
   (minus,Term,UNARYMINUS,Q6)

Q6=()
   (Q6a)

Q6a=(plus,Term,Add,Q6)
   (minus,Term,SUB,Q6)

Term=(Factor,Q5,SCALETERM)

Q5=()
   (mult,Factor,MPY,Q5)
   (div,Factor,DIVIDE,Q5)

Factor=(Difference,Q24)

Q24=()
   (power,Difference,RAISE,Q24)

Difference=(United,Q4)

Q4=()
   (differ,United,NEQ,Q4)

United=(Collation,Q3)

Q3=()
   (union,Collation,ORF,Q3)

Collation=(Tert,Q2)

Q2=()
   (mask,Tert,MSK,Q2)

Tert=(Sec,Q1)
   (Pri,Q1)

Q1=()
   (shift,SETSHIFT,Shifts,DOSHIFT,Q1)

Shifts=(Sec,PLUSSUB)
   (Tpri,SCALETERM,PLUSSUB)
   (orb,Subex,crb)

Sec=(Bitset,BITSIN,Pri,RHSBITS)

Bitset=(bits,CLEARTYPE,csb,int,Oneormore,FIELDPOSN,UNSFIELD,csb)

Oneormore=(NOBITS,NOSIG,PARTINT,comma,int)
   (ONEBIT)
```

Syntax 4 Primaries

```
Pri=(Tpri)
   (realcon,CONSTANT)
   (orb,STACKEXPR,SETPREF,Expr,crb,OFFSTEXPR)

Tpri=(intcon,CONSTANT)
   (id,LOOKUP,Q28)
   (CLEARTYPE,Type,TYPEXPR,orb,Expr,crb,EXPRTYPE)
   (NONAME,Singlesubs)
   (location,orb,Locvar,LOCACT,crb)
   (label,orb,id,Q32,crb)

Q28=(Subscript,VARCHECK)
   (RHSPTEST,Rhspsel)

Rhspsel=(VARCHECK)
   (Proccall)

Locvar=(id,LOOKUP,Q29)
   (NONAME,Singlesubs)

Q29=(Subscript)
   (AYMCHECK)

Q32=(LOOKUP,Singlesubs,LABSK)
   (LABL)

Singlesubs=(substart,csb,KILLEXPR)

Subscript=(Substart,Moresubs,csb,KILLEXPR)

Substart=(osb,SETUPSUB,Subex)

Moresubs=()
   (comma,SUBCOMMA,Subex,Moresubs)

Subex=(Subterms)
   (Cexpr,PLUSSUB)
   (Term,PLUSSUB,Subexcont)
   (zero)

Subexcont=()
   (Subterms)

Subterms=(plus,Term,PLUSSUB,Subexcont)
   (minus,Term,MINUSSUB,Subexcont)
```

Syntax 5 Conditions

```
Cexpr=(if,IFEX,Conditionlist,then,THENEX,Expr,else,ELSEFI,
        ELSEX,Expr,ELSEFI,FIEX)

Conditionlist=(Condition,Q31)

Q31=()
   (and,OUTCJ,Conditionlist)
   (or,ORACT,Conditionlist)

Condition=(Yesno,overflow,OVRTEST)
   (STACKEXPR,ANYPREF,Lcond,Rcond,RELATION)

Yesno=(no,SETNO)
   (SETYES,TYPEPROC)

Lcond=(Axpr)
   (zero,CONSTANT)

Rcond=(ro,Q30)
   (SETNEZ)

Q30=(Condterm,Condterms)
   (zero,ZEROCOMP)

Condterm=(Term,CONDPLUS)
   (plus,Term,CONDPLUS)
   (minus,Term,CONDMINUS)

Condterms=()
   (plus,Term,CONDADD,Condterms)
   (minus,Term,CONDSUB,Condterms)
```

Syntax 6 Assignment

```
S1=(St,Q39)

Q39=()
   (semi,Z1)

St=(S,ENDST)

S=()
   (id,Q38)
   (STATUSCHECK,Q26,A1)
   (Ifstat)
   (Forstat)
   (Gotostat)
   (Compound)
   (code,STATUSCODE,begin,Q35)
   (Answerstat)

Q38=(SETLAB,colon,S)
   (LOOKUP,STATUSCHECK,Q33)

Q33=(Q25,A1)
   (LHSPROC,Proccall,BEHEAD)

Q25=()
   (Subscript)

Q26=(Bitset,Lhsb)
   (NONAME,Singlesubs)

Lhsb=(id,LOOKUP,LHSBITS,Q25)
   (NONAME,LHSBITS,Singlesubs)

A1=(becomes,VARCHECK,SETASS,A2,STORE,BEHEAD)

A2=(A3)
   (Expr)

A3=(Cexpr)
   (string,STRINGEX)
   (zero,STOREZERO)
   (Term,ADDA,A4)
   (plus,Term,ADDA,A4)
   (minus,Term,SUBA,A4)

A4=(SIMPLEASS)
   (SELOPTA,A5)

A5=(Q6a)
   (Axpra)

Ifstat=(if,IFS,Conditionlist,then,ENDST,St,else,ELSES,S,FI)

Gotostat=(goto,id,Q27)

Q27=(LOOKUP,STATUSCHECK,Singlesubs,LABSK,GOTOSK)
   (GOTOL)
```

Syntax 7 Statements

```
Forstat=(for,STARTFOR,Locvar,CHECKCV,becomes,F1)

F1=(A3,ASSCV,F2)

F2=(F3)
   (while,Conditionlist,WHILEL,F3)
   (step,Expr,STEPUNT,until,Expr,STEPUNT,F3)

F3=(do,DOCS,S,ENDFOR)
   (comma,MOREFOR,F1)

Compound=(STATUSCHECK,begin,Q37)

Q37=(S1,end)
   (BEGINBLOCK,Body,end,ENDBLOCK)

Q35=(Cs1,end)
   (BEGINBLOCK,D1,ENDDECS,Cs1,end,ENDBLOCK)

Cs1=(Cs,Q34)

Q34=()
   (semi,Cs1)

Cs=()
   (Compound,STATUSCODE)
   (inst,INSTTYPE,Npart,OUTCODE)
   (id,CODELAB,colon,Cs)

Npart=(Nread)
   (Nwrite)
   (Nshift)
   (Njump)
   (Nloc)

Nread=(Nwrite)
   (Sgint,CONSTANT)
   (CLEARTYPE,Type,orb,Pnumber,SCALECON,crb,CONSTANT)

Nwrite=(anyacc,NISACC)
   (id,LOOKUPD,Nw1)
   (NONAME,Nw2)

Nw1=()
   (Nw2)

Nw2=(osb,Nw3,INCTOS,csb)

Nw3=(Sgint)
   (modacc,NISMOD,Zint)

Zint=(Sgint)
   (ZERONUM)

Nshift=(int,CODESHIFT)
   (modacc,Zint,CODESHIFT,NISMOD)

Njump=(id,LABL)
   (self,Zint,RELADD)

Nloc=(Nwrite)
   (string,STRINGEX)
```

Syntax 8 Table Declarations

```
Tabledec=(table,TYPEINT,TYPEARRAY,Newid,osb,int,TABLESIZE,csb,
                    Tabledetail,CLEARTYPE,TYPEINT)

Tabledetail=(osb,TABADD,Fieldlist)

Fieldlist=(CLEARTYPE,id,Fieldtype,TYPEARRAY,NEWNAME,Q60)

Q60=(csb)
   (semi,Q60a)

Q60a=(Q60)
   (Fieldlist)

Fieldtype=(Type,Sgint,FIELDDISP)
   (Partfield)
   (unsigned,Partfield,UNSFIELD)

Partfield=(orbosb,int,NOBITS,NOSIG,Intfixfield,crbcsb,
                        Sgint,FIELDDISP,commab,int,FIELDPOSN)

Intfixfield=(comma,Sgint,NOAFTER)
   (PARTINT)

Swlist=(id,SPECLAB,Q41)

Q41=()
   (comma,Swlist)

Overlaydec=(overlay,Base,with,OVERON,Datadec,OVEROFF)

Base=(id,LOOKUPD,Q43)
   (NONAME,osb,int,INCTOS,csb)

Q43=()
   (osb,Sgint,INCTOS,csb)

Datadec=(Type,Q21)
   (Tabledec)

Specialdec=(special,array,TYPEINT,TYPEARRAY,ADDRSPEC,Newid,becomes,Speclist)

Speclist=(Specitem,Q46)

Q46=()
   (comma,Speclist)

Specitem=(Zint,Q45)
   (string,SPECSTRING)
   (CLEARTYPE,Type,orb,Pnumber,SPECNUM,Morespec)

Morespec=(crb)
   (comma,Pnumber,SPECNUM,Morespec)

Q45=(SETTEN,Specadd)
   (CLEARTYPE,TYPEINT,SPECNUM)

Specadd=(colon,id,SPECLAB)
   (self,Sgint,SPECREL)
   (div,Q44)

Q44=(Sgint,NONAME,INCTOS,SPECCON)
   (Base,SPECCON)
```

Syntax 9 Procedure Declarations

```
Leveldec=(level,int,SETLEVEL)

Tracetype=(assignment,ASSTRACE)
   (loop,FORTRACE)
   (label,LABTRACE)
   (procedure,PROCTRACE)

Q48=(assignment,ASSTRACE,trace)
   (loop,FORTRACE,trace)
   (label,LABTRACE,trace)
   (procedure,Q49)

Q49=(Newid,BEGINPROC,Procrest)
   (PROCTRACE,trace)

Valproc=(value,TYPEVPROC,procedure)

Procrest=(Parameterpart,semi,Procbody,ENDPROC)

Parameterpart=()
   (orb,Parameters,crb)

Parameters=(Parameterset,Q65)

Q65=()
   (semi,NEXTPSET,Parameters)

Parameterset=(Type,Q63)
   (location,TYPELOC,Q62)
   (label,TYPELAB,Idlist)
   (switch,TYPEISWITCH,Idlist)
   (table,TYPEINT,TYPEIARRAY,Newid,osb,int,csb,PARAMTAB,Tabledetail,PARAMTAB)
   (value,Q64)
   (SETYES,TYPEPROC,procedure,Procparamlist)

Q64=(Type,Idlist)
   (TYPESPEC,Pairlist)
   (TYPEVPROC,procedure,Procparamlist)

Pairlist=(Newid,colon,Newid,Q61)

Q61=()
   (comma,Pairlist)

Q62=(Type,Idlist)
   (TYPESPEC,Pairlist)

Q63=(array,TYPEIARRAY,Idlist)
   (TYPETPROC,procedure,Procparamlist)

Procparamlist=(Newid,Paramlev2,Q58)

Q58=()
   (comma,Procparamlist)

Paramlev2=(BEGINPSPEC,Q57)

Q57=(orb,Paramtypelist,crb,ENDPSPEC)
   (ENDPSPEC)

Paramtypelist=(Paramtype,NEXTPARAM,Q56)

Q56=()
   (comma,NEXTPSET,Paramtypelist)
```

Syntax 10 Procedure Calls

```
Paramtype=(Type,Q54)
   (location,TYPELOC,Q53)
   (label,TYPELAB)
   (switch,TYPEISWITCH)
   (table,TYPEINT,TYPEIARRAY)
   (value,Q55)
   (SETYES,TYPEPROC,procedure)

Q55=(Type)
   (TYPESPEC,NEXTPARAM)
   (TYPEVPROC,procedure)

Q54=(array,TYPEIARRAY)
   (TYPETPROC,procedure)

Q53=(Type)
   (TYPESPEC,NEXTPARAM)

Procbody=(code,begin,KILLPARAMS,Od1,ENTERPROC,Cs1,end)
   (SETPARAMS,Q51)

Q51=(begin,Od1,PROCENTRY,S1,end,EXITCHECK)
   (ENDDECS,PROCENTRY,Ss,EXITCHECK)

Od1=(D1,ENDDECS)
   (ENDDECS)

Ss=()
   (Ifstat)
   (Forstat)
   (Answerstat)
   (Gotostat)
   (Id,LOOKUP,STATUSCHECK,Q33)
   (STATUSCHECK,Q26,A1)

Answerstat=(STATUSCHECK,answer,SETANS,Expr,ANSCHECK)

Proccall=(SETUPPROC,Selparams,CALLPROC,FINISHPROC)

Selparams=()
   (orb,Paramloop)

Paramloop=(NEXTPTYPE,Selptype,ANYMORE,Selmore)

Selptype;(Expr)
   (Locvar)
   (id,LOOKUP)
   (id,Q32)

Selmore=(Multitest)
   (comma,Paramloop)

Multitest=(crb)
   (comma,CALLPROC,MULTICALL,Paramloop)
```

Syntax 11 Common Communicators

```
Commondec=(common,orb,COMON,Commonlist,COMOFF,crb)

Commonlist=(STARTDEC,Commonitem,Q23)

Q23=()
   (semi,Commonlist)

Commonitem=()
   (Type,Q22)
   (Valproc,Comprocs)
   (Tabledec,Presetlist)
   (Overlaydec)
   (Specialdec)
   (label,TYPELAB,Comspecidlist)
   (switch,TYPEISWITCH,Comspecidlist)
   (SETYES,TYPEPROC,procedure,Comprocs)

Q22=(Q21,Presetlist)
   (TYPETPROC,procedure,Comprocs)

Comspecidlist=(Comspecid,Comspecidcont)

Comspecidcont=()
   (comma,Comspecidlist)

Comspecid=(ADDRSPEC,Newid,SPECONE)

Comprocs=(Comspecid,Paramlev2,Q20)

Q20=()
  (comma,Comprocs)

Abslist=(CLEARTYPE,Absitem,Q11)

Q11=()
   (semi,Abslist)

Absitem=()
   (Type,Q10)
   (Valproc,Absprocs)
   (label,TYPELAB,Absidlist)
   (switch,TYPESWITCH,Absidlist)
   (table,TYPEINT,TYPEARRAY,Absid,osb,int,csb,Tabledetail)
   (SETYES,TYPEPROC,procedure,Absprocs)

Q10=(Q9,Absidlist)
   (TYPETPROC,procedure,Absprocs)

Q9=()
   (array,TYPEARRAY)

Absidlist=(Absid,Q7)

Q7=()
   (comma,Absidlist)

Absid=(id,div,int,ABSADD,NEWNAME)

Absprocs=(Absid,Paramlev2,Q8)

Q8=()
   (comma,Absprocs)
```

```
Q69=(orb,Liblist,crb)
   (CLEARTYPE,Q66)

Q66=(Type,Q67)
   (Valproc,Libid,SETLIBSEG,colon,Newid,NEXTPSET,Procrest,FINISHSEG)
   (SETYES,TYPEPROC,procedure,Libid,SETLIBSEG,Procrest,FINISHSEG)

Q67=(Libid,becomes,Pnumber,SCALECON,SETLIBVAR)
   (TYPETPROC,procedure,Libid,SETLIBSEG,Procrest,FINISHSEG)

Liblist=(CLEARTYPE,Libitem,Q19)

Q19=()
   (semi,Liblist)

Libitem=()
   (Type,Q18)
   (Valproc,Libprocs)
   (SETYES,TYPEPROC,procedure,Libprocs)

Q18=(Libidlist)
   (TYPETPROC,procedure,Libprocs)

Libidlist=(Libid,Q16)

Q16=()
   (comma,Libidlist)

Libid=(id,div,int,LIBADD,NEWNAME)

Libprocs=(Libid,Paramlev2,Q17)

Q17=()
   (comma,Libprocs)

Extlist=(CLEARTYPE,Extitem,Q15)

Q15=()
   (semi,Extlist)

Extitem=()
   (Type,Q14)
   (Valproc,Extprocs)
   (label,TYPELAB,Extidlist)
   (switch,TYPESWITCH,Extidlist)
   (table,TYPEINT,TYPEARRAY,Extid,osb,int,csb,Tabledetail)
   (SETYES,TYPEPROC,procedure,Extprocs)

Q14=(Q9,Extidlist)
   (TYPETPROC,procedure,Extprocs)

Extidlist=(Extid,Q12)

Q12=()
   (comma,Extidlist)

Extid=(id,div,SETTEN,int,Zint,EXTADD,NEWNAME)

Extprocs=(Extid,Paramlev2,Q13)

Q13=()
   (comma,Extprocs)
```

'INTEGER' 'PROCEDURE' ACCOF

('VALUE' 'INTEGER' REF);

This procedure is used to find out whether the Arithmetic Operand referred to by REF, is held in an accumulator. If it is, then the result is the number of the accumulator, otherwise zero.

'PROCEDURE' ACCPICK

('VALUE' 'INTEGER' REF,ACC);

This procedure copies the Arithmetic Operand referred to by REF into the specified accumulator ACC. If it already exists in that accumulator, then no action is taken; but if the accumulator is occupied by another operand, then this is first stored. This procedure is used exclusively with floating point operations.

'PROCEDURE' ACCUPDATE

('VALUE' 'INTEGER' REF);

If the Arithmetic Operand on the stack, referred to by REF is not a constant, then a check is made to see if either the direct or indirect address is an accumulator.

If one is, then the appropriate entry is made in ACCS to keep it up to date.

'INTEGER' 'PROCEDURE' ADDADD

('VALUE' 'INTEGER' ADD,INC);

This procedure adds an increment INC, to the index field of an address ADD, ensuring that any carry from the index field (14 bit) does not overflow into the other fields.

'PROCEDURE' ADDRARRAY

('VALUE' 'INTEGER' START);

This procedure is called to insert the addresses into the identifier records of one or more arrays declared with the same bounds. It is called from ONEDIM in the case of single dimensional arrays, and from FIRSTDIM in the case of multi-dimensional arrays. The procedure operates by calling DODIM with the local procedure MINE as the procedure parameter. This procedure parameter is applied to each identifier record, in the reverse order in which the identifiers were declared. Thus the parameters supplied to DODIM have to ensure that the highest address is supplied to MINE first, and the lowest last. The number of identifier records to be addressed is given in ARRAYS.

'PROCEDURE' ADDSUB

('VALUE' 'INTEGER' FUN);

This procedure is called from ADD with a value of FUN of 2, and from SUB with a value of FUN of 3. This procedure generates the instructions required to perform addition and subtraction operations. The scale to which this will be carried out is given by EXPSCALE. The left hand operand is copied into an accumulator, unpacked, and rescaled as required by means of a call of PICK with GOODACC providing the accumulator number. If the right hand operand is a constant then it is rescaled if required. Otherwise it is copied into a second accumulator if it requires unpacking or rescaling. The instruction with the function code FUN is then output. The top operand record is removed from the stack and the type of the other operand record changed to that given by EXPSCALE.

'PROCEDURE' ANSLINK;

This procedure is called in the case where an 'ANSWER' statement is not the last statement of a procedure. If the use of the procedure trace facility is not required, an exit instruction is output by means of OUT27, and the fact recorded by setting EXITCH to one. If however trace is required, an unconditional jump to the end of the procedure, where the call of L6 is made, is output. This jump is chained up using EXITCH as the address, which is then updated.

'INTEGER' 'PROCEDURE' ANYMORE;

This procedure is a selector action called directly from the syntax, after each parameter of a procedure call. If the procedure requires further parameters, a result of 1 is returned, otherwise 0. Before returning to the syntax analyser, however, a check is made to ensure that the parameter just processed is of the correct type.

If the parameter required is an untyped 'VALUE' or 'LOCATION', the type of the actual parameter is copied into EXPSCALE. A switch is then made, using PARAMCLASS, to check the type of the parameter. In the case of 'VALUE' parameters, if the type of the actual parameter is not that of the formal parameter, the type of the actual parameter is changed to that required. If the parameter is a constant, then this is simply rescaled. Otherwise the parameter is evaluated and rescaled, the result being left in an accumulator, which will be the one required for passing the parameter, unless it is already in use.

In the case of 'LOCATION' parameters a check is made to ensure that the actual parameter is not a partword or array, and is of exactly the required type and scale.

In the case of 'ARRAY' and 'SWITCH' parameters a check is made that they are of the exact type required.

If the actual parameter is any type of procedure then a check is made that it is not being used in a potentially recursive situation. This is detected by examining the sign bit of its PARAMSPEC, which will be set if the procedure

'INTEGER' 'PROCEDURE' ANYMORE

is not yet complete, i.e. the use of the procedure identifier occurs within the procedure body.

A check is then made to ensure that the procedure supplied is allowable. If the formal parameter is an untyped procedure then the actual parameter may be any type of procedure except a 'VALUE' 'PROCEDURE', otherwise the actual parameter must be exactly of the right type. No check is made that its parameter requirements match, as insufficient information is kept to enable this to be done.

If the formal parameter is a 'LABEL' then no checking is required, as it is done elsewhere.

After the type checking has been completed, the parameter Arithmetic Operand is processed by LOCACT if a location is required to be passed. If the parameter has a non zero indirect address then the parameter is copied into an accumulator. The parameter specification pointer, PSP, is incremented. If the next parameter is the second, type, parameter of a non-standard parameter (pair) then a call is made on MAKEPARAM to set up an Arithmetic Operand record for this second parameter. A flowchart is given for this procedure.

'INTEGER' 'PROCEDURE' ARRAYBASE;

The result of this procedure is the address of first location of the (last dimension of the) array currently being declared. In the case of a non-overlay declaration this is given by the value of DATAMAX, or in the case of an overlay declaration, OVERBASE. At this point the appropriate variable has not yet been incremented by the total data requirement of the array.

'PROCEDURE' BEGINBLOCK;

This procedure is called at the start of a block and performs the housekeeping associated with the block structure. It is called directly from the syntax for blocks within a segment. It is also called from PROCSTACK to make up the body of a procedure a block, even if it has no declarations at its head, and also from BEGINPROG.

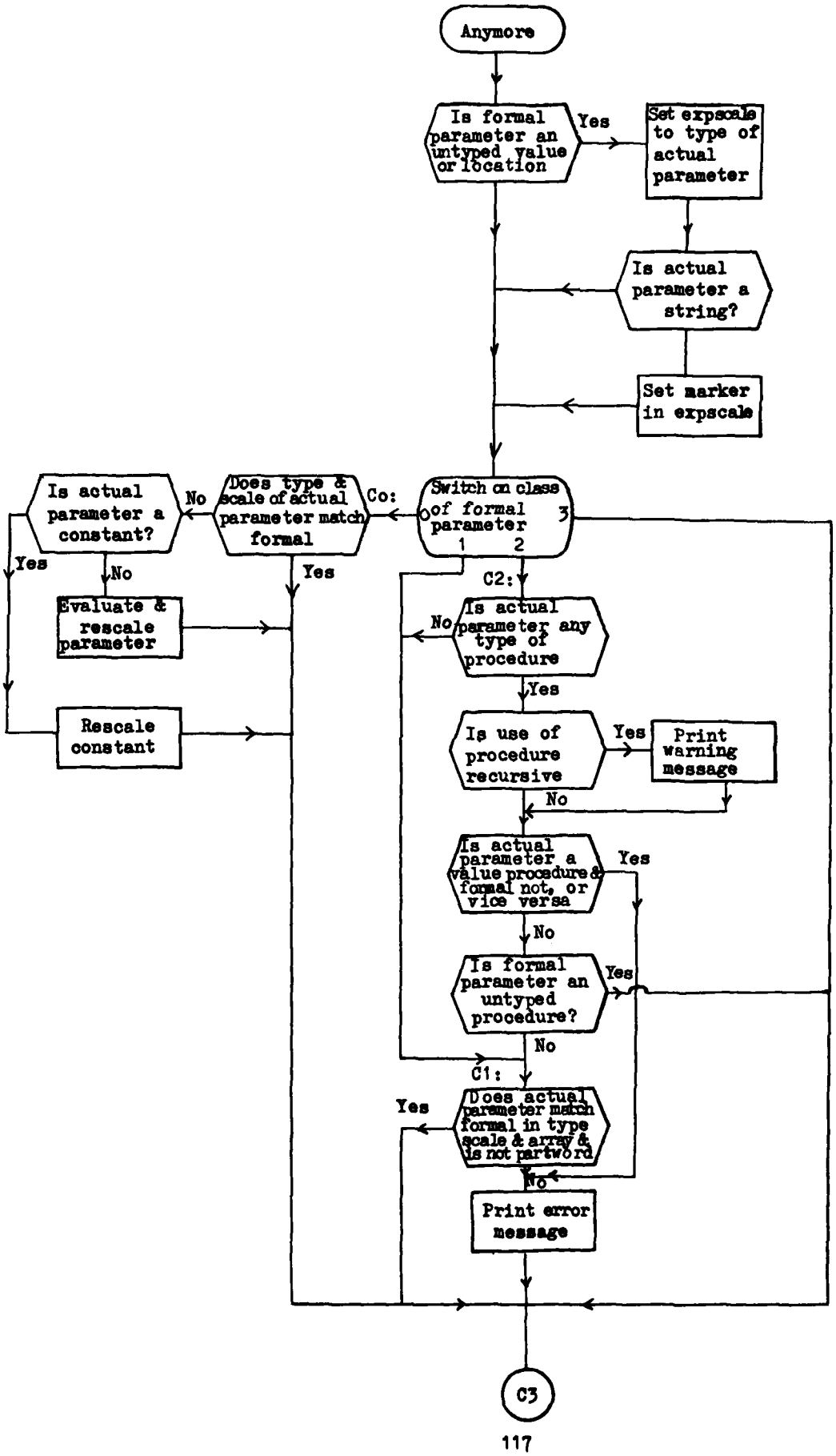
This procedure starts with a call of BEGLABBLOCK to set up a block level for labels. This is followed by a call of ONSTACK to place on the stack the values of the following variables whose values will be reset at the end of the block:

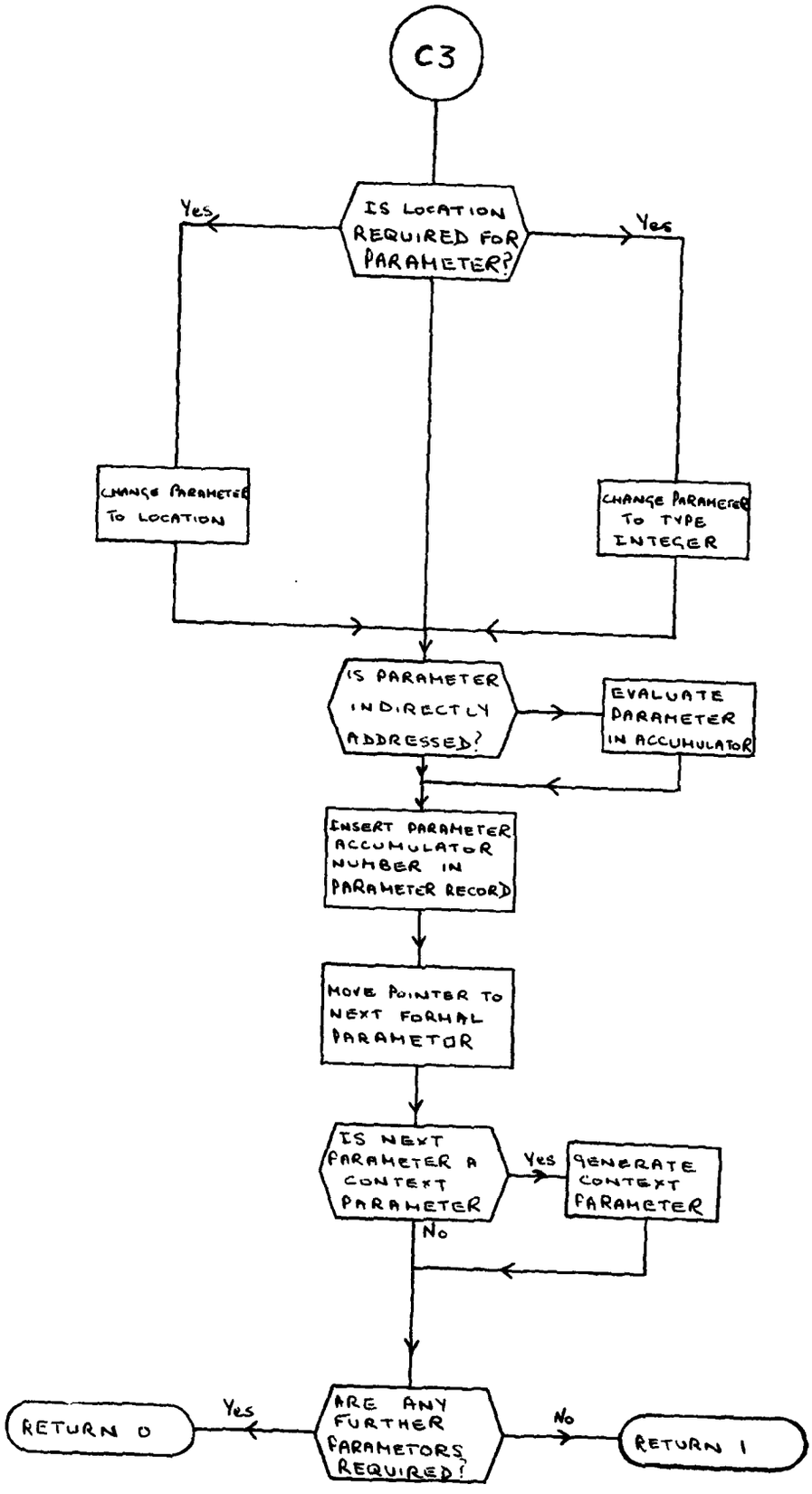
BLOCKCHAIN,LEVEL,TRACE,LOCALLIMIT,DATASTART,DATAPTR,DATAMAX, and PRESETOK.

The value of LOCALLIMIT is set to the value of DECLIST at the start of the block. The variable DATAMAX will be used to allocate workspace to the declarations following, and is set to the value of DATASTART of the outer block at this point. (See also ENDBLOCK)

'PROCEDURE' BEGLABBLOCK;

This procedure begins a block as far as the administration of labels is


Anymore
Is formal parameter an untyped value or location
Yes
Set expscale to type of actual parameter
Is actual parameter a string?
Set marker in expscale
Switch on class of formal parameter
1
2
3
Co:
Does type & scale of actual parameter match formal
No
Is actual parameter a constant?
Yes
No
Evaluate & rescale parameter
Rescale constant
Yes
C2:
Is actual parameter any type of procedure
No
Yes
Is use of procedure recursive
Yes
Print warning message
No
Is actual parameter a value procedure & formal not, or vice versa
Yes
No
Is formal parameter an untyped procedure?
Yes
No
C1:
Does actual parameter match formal in type scale & array &, is not partword
Yes
No
Print error message
C3


C3
IS LOCATION REQUIRED FOR PARAMETER?
Yes
Yes
CHANGE PARAMETER TO LOCATION
CHANGE PARAMETER TO TYPE INTEGER
IS PARAMETER INDIRECTLY ADDRESSED?
EVALUATE PARAMETER IN ACCUMULATOR
INSERT PARAMETER ACCUMULATOR NUMBER IN PARAMETER RECORD
MOVE POINTER TO NEXT FORMAL PARAMETOR
IS NEXT PARAMETER A CONTEXT PARAMETER
Yes
GENERATE CONTEXT PARAMETER
No
ARE ANY FURTHER PARAMETORS REQUIRED?
Yes
RETURN 0
No
RETURN 1

'PROCEDURE' BEGLABLOCK

concerned. It is called from BEGINBLOCK at the start of a normal block, and also from STARTFOR to prevent the possibility of a jump being made to a label within a 'FOR' statement from a point outside. The current values of LABDECLIST and LABSTACKPTR are stacked using a LABCHAIN. LABDECLIST is then set to zero, thus labels set or referred to in the enclosing block are rendered inaccessable.

'PROCEDURE' BEHEAD;

This procedure removes the top Arithmetic Operand from the stack and deletes any reference to it in ACCS. LH and RH are reset. This procedure is called from the syntax, and from other compiling actions.

'PROCEDURE' CALLLIB

('VALUE' 'INTEGER' LNO,SPIEL);

This procedure is used to set up a call of the Library Procedure whose reference number is given by LNO. This procedure is used by the compiler to set up calls on tracing and floating point arithmetic procedures, it is not used to set up calls of Library Procedures explicitly referred to. The procedure forms the required address by adding an L tag to the LNO and calling OUT27.

'INTEGER' 'PROCEDURE' COPYINACC

('VALUE' 'INTEGER' REF);

This procedure is used to find out whether the Arithmetic Operand referred to by REF is in an accumulator, or if there is a suitable copy of the operand in an accumulator. If the Operand is a constant, or is a location, or is indirectly addressed then the result is zero, otherwise a call is made on FINDOUT and its result taken.

'PROCEDURE' DIAG

('VALUE' 'INTEGER' TA,TB);

This procedure is used to output, on the printer, a commentary of important points reached during compilation. The current program address PTA is printed out, followed by the two strings TA and TB separated by " : ". This is used to indicate the positions of labels and the starting point of procedures and loops. If the value of the variable LEVEL indicates that this information is required at load time, then the string TB is output as part of the relocatable binary output, together with the appropriate directive tag (35).

'INTEGER' 'PROCEDURE' DIRINDADD

('VALUE' 'INTEGER' REF);

The result of this procedure is the direct address of the Arithmetic Operand referred to by REF. If however this is null then the result is the indirect address. This procedure is used where it is known that one address has a value, and the other is null. A similar action takes place in LOOKUPD.

```
'PROCEDURE' DODIM

          ('VALUE' 'INTEGER' START,INC;
           'PROCEDURE' ACT('VALUE' 'INTEGER'));
```

This procedure is used for processing array declarations. When called at the first (or only) dimension, the number of array identifiers declared with the same bounds is given by ARRAYS. If there is only one dimension then DODIM is called by ADDRARRAY which is called by ONEDIM. The effect is to proceed down the list of identifier records inserting the direct address. The total store requirement for the arrays is calculated by ARRAYS * NUMBER.

If however there is more than one dimension, each dimension except the last, requires the setting up of Iliffe vectors in Special Data. In this case the first dimension causes FIRSTDIM to call ADDRARRAY and hence DODIM to insert the addresses of the first level of the Iliffe vectors into the identifier records. If there are three or more dimensions then further levels of Iliffe vectors are set up by MIDDIM calling DODIM with OUT1014CS as the parameter ACT. This outputs all but the last level of vector. In this case the vectors point at the next level vectors. For the last (or second) dimension LASTDIM calls DODIM, again with OUT1014CS as a parameter, to output the highest level of vector. This level point at the actual data area occupied by the array data. Note that even if the arrays are declared as part of an overlay declaration separate vectors are set up as required.

```
'PROCEDURE' DOSTRING

          ('VALUE' 'INTEGER' STRING;
           'PROCEDURE' PROC('VALUE' 'INTEGER') );
```

This procedure applies the procedure PROC to each word in turn of the string STRING. The parameter PROC will be either OUT24CS or OUTCONT.

```
'PROCEDURE' DUEERR;
```

This procedure is called in the case where a statement is not labelled, and not directly entered from the previous statement. If in this case DUECHAIN is null, the statement cannot be (legally) entered, and a message is output to this effect.

```
'PROCEDURE' DUMMY;
```

This procedure does nothing. It is used as a dummy parameter in a call of STATUSTEST by STATUSCODE.

```
'PROCEDURE' DUMPACC

          ('VALUE' 'INTEGER' A);
```

This procedure outputs an instruction to dump the contents of the specified accumulator A in an anonymous temporary workspace, allocating this as required. The direct address of the relevant Arithmetic Operand is set to the address of the workspace and flagged as being temporary. The entry in ACCS for the accumulator A is cleared. This procedure is called when it is required to preserve an Arithmetic Operand which exists only in an accumulator.

'PROCEDURE' DUMPACCS;

This procedure outputs instructions to store in anonymous temporary workspace all Arithmetic Operands which exist only in accumulators.

'PROCEDURE' ENDBLOCK;

This procedure is called at the end of a block and performs the housekeeping associated with the block structure. It is called directly from the syntax for blocks within a segment. It is also called from ENDPROC at the end of a procedure, and also from ENDPROG.

The value of DATAMAX at the end of the block is temporarily stored in MAX, and DECLIST reset to its value at the start of the block, this value being held in LOCALLIMIT. OFFSTACK is then called to reset the values of BLOCKCHAIN, LEVEL,TRACE,LOCALLIMIT,DATASTART,DATAPTR,DATAMAX, and PRESETOK. This also has the effect of removing from the stack the identifier records set up inside the block, as these identifiers are now out of scope. If the value of MAX, i.e. the highest numbered data space allocated within the block, exceeds the restored value of DATAMAX, i.e. the highest numbered data space allocated so far, then DATAMAX is set to this larger value. Finally a call is made on ENDLABBLOCK to compact the label stack and remove any labels now out of scope.

'PROCEDURE' ENDLABBLOCK;

This procedure ends the compilation of a block as far as the administration of label records is concerned. It is called from ENDBLOCK and ENDFOR. The previous values of LABDECLIST and LABSTACKPTR are unstacked from the main stack, and the list of label records previously referred to by LABDECLIST processed. These may be divided into three categories:

a) Labels set in the inner block.
b) Labels unset in the inner block and referred to in the outer block.
c) Labels unset in the inner block and not referred to in the outer block.

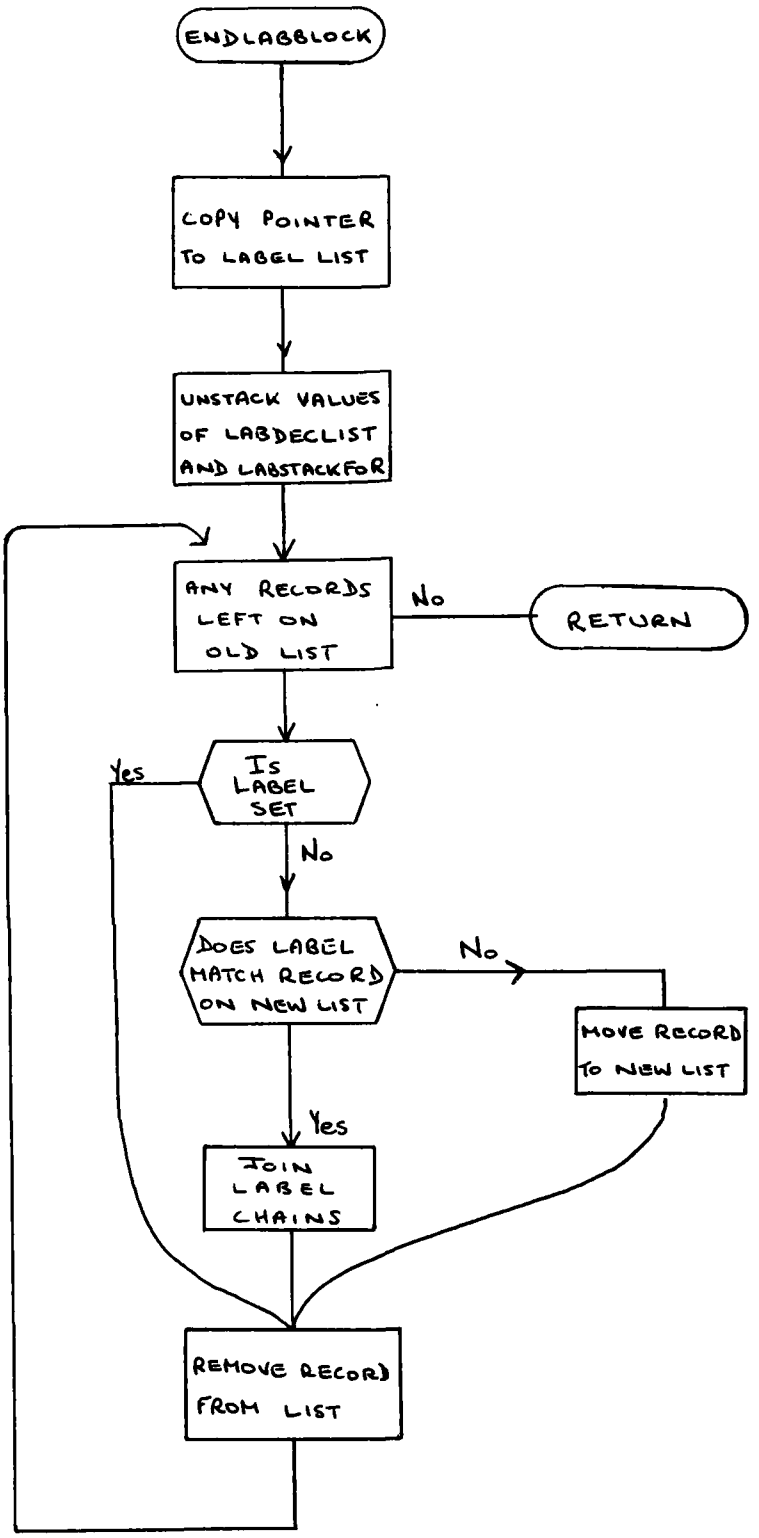
In the case of (a) the record is simply discarded, and in the case of (b) the chains are joined with JOINCHAINS before the record is discarded. In the case of (c) the record is moved up, and joined on to the label list of the outer block. A flowchart is given for this procedure.

'PROCEDURE' ENDSEG;

This procedure is called at the end of a segment. It outputs an end of segment tag(39) with the checksum for the characters forming the segment. This is then followed by a tail of blank, a stop directive, erases and blank by means of a call of TTAIL. The variable TEST is set to zero in case it had been set by a 'TEST' directive, or a call of STOPOP.

'PROCEDURE' ENTERPROC;

This procedure is called when the entry point of a procedure is reached. A call is made on SETDUE to set the entry address of the procedure. In the case of a Library Procedure this address is set into the L area by the loader,


ENDLABBLOCK
COPY POINTER TO LABEL LIST
UNSTACK VALUES OF LABDECLIST AND LABSTACKFOR
ANY RECORDS LEFT ON OLD LIST
No
RETURN
Yes
IS LABEL SET
No
DOES LABEL MATCH RECORD ON NEW LIST
No
MOVE RECORD TO NEW LIST
Yes
JOIN LABEL CHAINS
REMOVE RECORD FROM LIST

'PROCEDURE' ENTERPROC

clearing the unsatisfied reference, and in the case of a procedure within a segment this address is set into the location allocated within the Special Data area. STATUS is set to 0 to indicate that control is passed to the first statement of the procedure.

'PROCEDURE' EXCPSPEC;

This procedure exchanges the values of IDTYPE and IDTYPE2, and also PARAMPTR and PARAMPTR2. This action is required in the case of the parameter specifications of a 'PROCEDURE' parameter in the heading of a procedure declaration. As the action of the procedure is symetrical, separate procedures are not required to store and restore the values of IDTYPE and PARAMPTR.

'INTEGER' 'PROCEDURE' FINDACC

('VALUE' 'INTEGER' A);

This is the main accumulator allocation algorithm for finding free accumulators. Starting with the accumulator specified by the parameter A, a scan is made, with a decreasing index of the array ACCS to find a suitable accumulator. A mapping function is applied to each entry of ACCS to produce a 24 bit "value" for that accumulator. The accumulator with the least value, or the higher numbered one of equal values, is chosen. The mapping function is simple and arranges that accumulators are chosen in the following order:

a) Free accumulators
b) Accumulators containing a copy of a stored word
c) Accumulators containing (the only copy) an Arithmetic Operand.

If an accumulator of category (c) is chosen, its contents are dumped using DUMPACC. Within this category the accumulator holding the Arithmetic Operand furthest down the stack will be chosen. Within category (b) data local to the segment is regarded as having a greater "value" than data external to it; and within a segment, data having a higher address (i.e. declared nearest) is regarded as having a higher "value".

The result of the procedure is the number of the chosen accumulator.

'INTEGER' 'PROCEDURE' FINDOUT

('VALUE' 'INTEGER' ADD);

This procedure is used to find whether the address given by the parameter ADD is an accumulator address or if there is a copy of the word already in an accumulator. If the address is an accumulator address, the result is the number of the accumulator. Otherwise a search is made through ACCS, starting with the lowest numbered accumulator. If the entry in ACCS matches ADD then the result is the accumulator number. If no match can be found the result is zero. As a safeguard a zero result is returned for a zero address.

'INTEGER' 'PROCEDURE' FINDPREF;

This procedure is used to find a free accumulator. If the preferred accumulator for the expression, whose number is given by PREFACC, is free then this one is taken. Otherwise a call is made on FINDACC.

'PROCEDURE' FINISHPSPEC;

This procedure is called to complete the parameter specification record of a procedure identifier record on the stack. Any unused space in the parameter specification record is handed back to the stack, and the anti-recursion marker in the PARAMSPEC field deleted.

'PROCEDURE' FINISHSEG;

This procedure is called from ENDPROG at the end of a program segment, and from COMOFF at the end of a 'COMMON' segment. It is also called from the syntax at the end of a 'LIBRARY' 'PROCEDURE' segment. If any labels remain unset in program or library segments, (LABDECLIST is not used in 'COMMON') then a warning is printed and the labels set to calls of L2 with the label identifier string as a parameter. ENDSEG is called to output the end of segment directive and tail on the tape. This is then followed by the "size block" output on tape, giving the size of the Program,Data, and Special areas; this information also being printed. A tail is punched on the tape, and "END OF SEGMENT" printed.

'PROCEDURE' FIXCON;

If the operand on the top of the stack is an integer constant, then this is converted to a fixed constant with the minimum number of integer bits required to hold the number without loss of significant bits. This number is given by SGB-1.

'PROCEDURE' FIXLABEL

('VALUE' 'INTEGER' LAB);

This procedure is used when the label specified by the pointer LAB is to be set to the current program address. If the label record indicates that it has been previously set then a call is made on STOPOP to print a message. Otherwise calls are made on SETCHAINTOPTA to set the program area and (special) data area chains, if any. The two addresses in the label record are set to the current program address and marked (sign bit set) to indicate that the label is now set. A call is made on ZEROACCS to clear the accumulator record ACCS, even though this might also be called by SETCHAINTOPTA. Finally a call is made to DIAG to print out the label identifier and the current program address. FIXLABEL is called from SETLABEL to set labels normally, from FINISHSEG to set labels unset in a segment, and from CODELAB to set labels in code.

'PROCEDURE' FLOATIT

('VALUE' 'INTEGER' REF)

This procedure ensures that the Arithmetic Operand referred to by REF is of type floating. If the operand is a constant then it is rescaled using SCALENUM. Otherwise if a conversion is required, a call is set up to L10 to float the value.

'PROCEDURE' FLOATOP

('VALUE' 'INTEGER' OP);

This procedure outout the instructions to load the operands into @7 and @6, and to call the appropriate Library procedure. If the right hand operand is already in @7, then a procedure which expects the operands to be reversed is called. A table is given below showing the relation between the operators, the value of OP, and the Library procedure called.

Operator	OP	Normal	Reversed
*	1	L11	L11
+	2	L12	L12
-	3	L13	L17
/	4	L14	L18
↑	5	L15	L19
↑ (integer)	6	L16	L20

After the instructions have been output, the top operand record is removed from the stack, the type of the other operand record changed to floating, and the accumulator record updated.

'PROCEDURE' FORCOM;

This procedure is called at the end of each element of a forlist containing more than one element. After masking the bottom two bits, FORSTATE will be zero for a simple expression, and two for a step-until; otherwise the element is a while.

If the element is a step-until a call is made on FORTEST to output the instructions to test the control variable against the limit. A call of the controlled statement, as an anonymous procedure, is then output. If the element is a step-until, a call is then made on FORINC to output instructions to increment the control variable. Unless the element is of the form Expression, a call is then made on UJBACK to output an unconditional jump back to the step-until or while test. FORSTATE is then set to four prior to processing the next element.

'PROCEDURE' FORINC;

This procedure outputs the code to increment the control variable in a step-until element of a For statement. ASSFUN is set to OCTAL(12) and a call made on STOREAWAY. This causes instructions to be output which copy the step value into an accumulator and add into store.

'INTEGER' 'PROCEDURE' FORMMASK

('VALUE' 'INTEGER' REF);

This procedure is used to form the masks (bit patterns) required for the packing and unpacking in part word operations. The parameter REF is a reference to an Arithmetic Operand on the stack, whose PARTWORD field specifies the location of the required part word within the whole word. This information is obtained from the two fields LSS and MSS of PARTWORD. LSS specifies the number of shifts required to align the field with the least significant end of the word, and MSS the number of shifts required to align the field with the most significant end. Thus for whole word fields these are both zero.

The result of the procedure FORMMASK is a bit pattern consisting of all ones in bit positions corresponding to the specified field, and all zeros elsewhere.

'PROCEDURE' FORTEST;

This procedure outputs the instructions for the testing of the control variable against the limit value of a step-until element of a for statement. This test is always performed using accumulator 7, and the test instruction is always a jump-if-negative; this jump taking place when the element is exhausted. This procedure is used in all step-until cases except where the for statement has only one element, which is a step-until with all three expressions being constants.

If the step is a negative constant, an instruction is output to copy the control variable into accumulator 7; otherwise an instruction is output to copy it negatively. Provided that the limit is not a zero constant, a further instruction is output; if the step is a negative constant, this subtracts the limit value, otherwise it adds it. If the step is not a constant, an instruction is output to multiply the result by the step value. The test instruction is then output, its address being recorded in SKIPCHAIN. Finally the Arithmetic Operand for the limit is deleted from the stack. The code generated is summarised below:

If limit is not zero constant	
If step not a constant	(Limit-Controlvar)*Step
If step a +ve constant	+Limit-Controlvar
If step a -ve constant	+Controlvar-Limit
If limit a zero constant	
If step not a constant	-Controlvar*Step
If step a +ve constant	-Controlvar
If step a -ve constant	+Controlvar

'PROCEDURE' GIVEUP

('VALUE' 'INTEGER' S);

This procedure is called in case of irrecoverable errors. The contents of the input buffer are printed using PRINTBUFF. This is then followed by the characters "FAILED : " and the string S. The procedure does not return control to the point of call but jumps to the label STARTUP to re-initialise the compiler.

'INTEGER' 'PROCEDURE' GOODACC

('VALUE' 'INTEGER' REF);

This procedure choses a suitable accumulator for use with the Arithmetic Operand referred to by REF. If the operand exists in an accumulator, then this is chosen. Otherwise a call is made to FINDPREF to find a free accumulator.

'INTEGER' 'PROCEDURE' GOODONE

('VALUE' 'INTEGER' REF,LIM);

This procedure choses a suitable accumulator number for the evaluation of a subscript expression incorporating the Arithmetic Operand referred to by REF, which is less than or equal to the value LIM. A detailed flowchart is given.

'PROCEDURE' GOODPICK

('VALUE' 'INTEGER' REF,NEG);

This procedure loads the Arithmetic Operand referred to by REF, into a suitable accumulator. The operand is not rescaled but unpacked if a partword. If NEG is non zero then the operand is negated.

'INTEGER' 'PROCEDURE' GRABSTACK

('VALUE' 'INTEGER' N);

This procedure allocates the number of words given by N, of workspace on the compiler's main stack. The result is the starting point of this space. A call is made on STACKCHECK to test for stack collision.

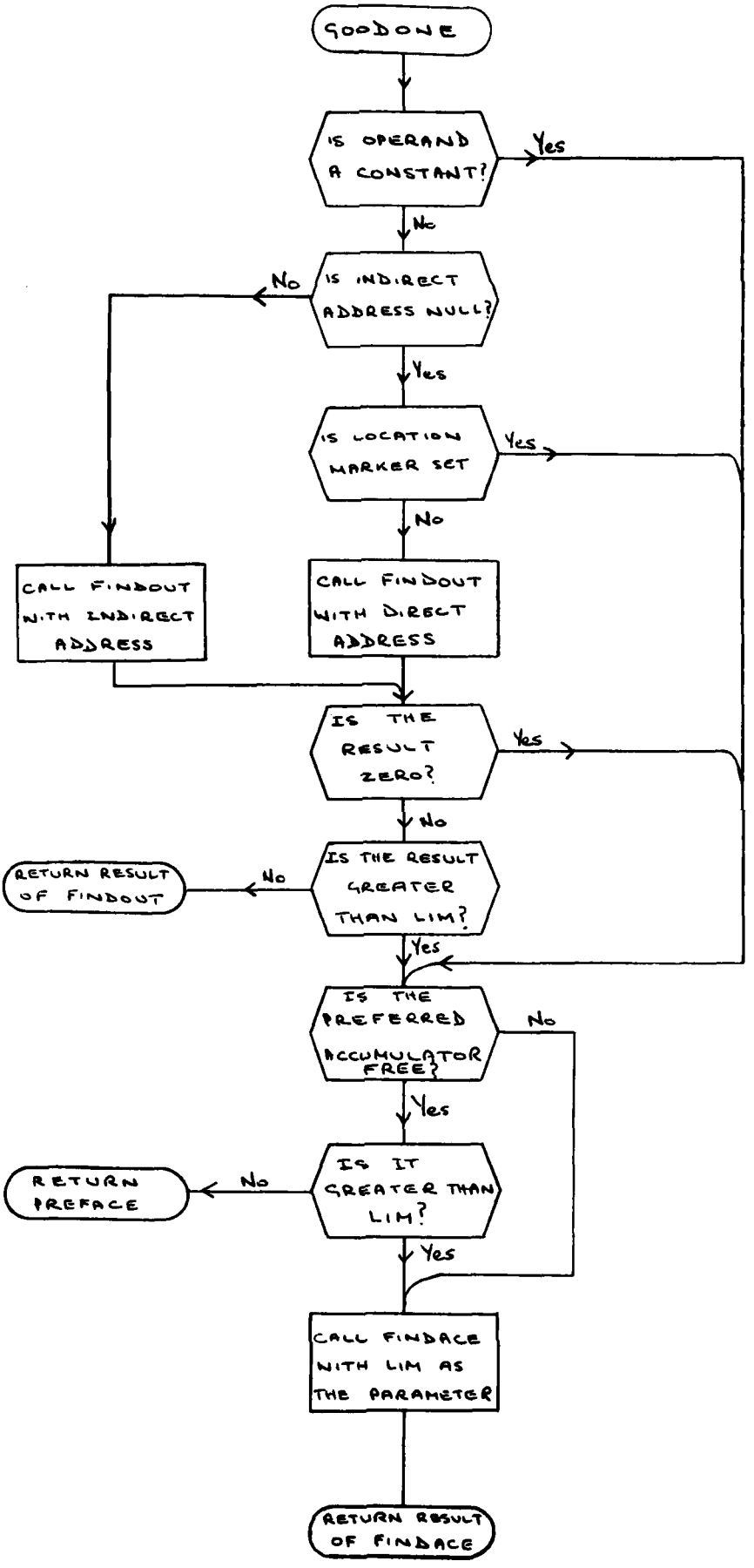
'PROCEDURE' GRABVAR;

This procedure obtains five words on the top of the stack to form a new Arithmetic Operand. This area is initialised by setting the PARTWORD,DIRADD, and INDADD words to zero.

'PROCEDURE' HALT

('VALUE' 'INTEGER' STRING);

This procedure outputs the characters "HALTED - ", the string STRING, and a newline. It then waits for Handswitch 0 to be moved from 0 to 1.


GOODONE
IS OPERAND A CONSTANT?
Yes
No
IS INDIRECT ADDRESS NULL?
No
Yes
IS LOCATION MARKER SET
Yes
No
CALL FINDOUT WITH INDIRECT ADDRESS
CALL FINDOUT WITH DIRECT ADDRESS
IS THE RESULT ZERO?
Yes
No
RETURN RESULT OF FINDOUT
No
IS THE RESULT GREATER THAN LIM?
Yes
IS THE PREFERRED ACCUMULATOR FREE?
No
Yes
RETURN PREFACE
No
IS IT GREATER THAN LIM?
Yes
CALL FINDACE WITH LIM AS THE PARAMETER
RETURN RESULT OF FINDACE

'PROCEDURE' INACC

('VALUE' 'INTEGER' ACC,REF);

This procedure updates the Arithmetic Operand on the stack whose reference is REF, to show that it is now held in the accumulator whose number is given by ACC. A call of ACCUPDATE is made to update ACCS.

'PROCEDURE' INST

('VALUE' 'INTEGER' REF,ACC,FUN);

This is the general procedure for the output of instructions whose address and modifier parts are derived from an Arithmetic Operand record on the stack. The required accumulator and function number are passed as ACC and FUN, and the parameter REF refers to an Arithmetic Operand. The required modifier is obtained by the use of ISMOD. The address is obtained from the direct address, but in certain cases an intermediate transformation is required. Where the Arithmetic Operand is a constant, this is output by OUTWCONST and the address supplied by it (W_0) used. Where the address is a label address, *LBM is set, and USELAB is called to obtain the actual or the chain address.* Where a location is required, LCM will be set, and four is added to the function code. In this case the function code will always be less than four.

'INTEGER' 'PROCEDURE' INSTTYPE;

This procedure is a selector action called directly from the syntax analyser, after the three letter function in code. The procedure returns a number in the range 0 to 4 as follows:

Function	Type
LDX	0
NLX	0
ADD	0
SUB	0
LOC	4
LAB*	3
STO	1
STN	1
ADS	1
SSB	1
EXC	1
AND	0
NEQ	0
ORF	0
JZE	3
JNZ	3
JGE	3
JLT	3
OVR	3
JBS	3
OUT	1
JCS	1

'INTEGER' 'PROCEDURE' INSTTYPE

Function	Type
SRA	2
SLA	2
SRL	2
SLC	2
SLL	2
SLV	1
MPY	0
DIV	0

Note: LAB is initially given the function code 05, to differentiate it from LOC. This procedure changes the code to 04.

'INTEGER' 'PROCEDURE' ISINACC

('VALUE' 'INTEGER' REF);

This procedure is used to find out whether the Arithmetic Operand referred to by REF is in an accumulator or if there is a suitable copy of the operand in an accumulator. If there is the both the Arithmetic Operand and the accumulator record,ACCS, are updated. A call is made on COPYINACC whose result is made the result of ISINACC. If this is non zero, a call is made on INACC to update the record.

'PROCEDURE' ISMOD

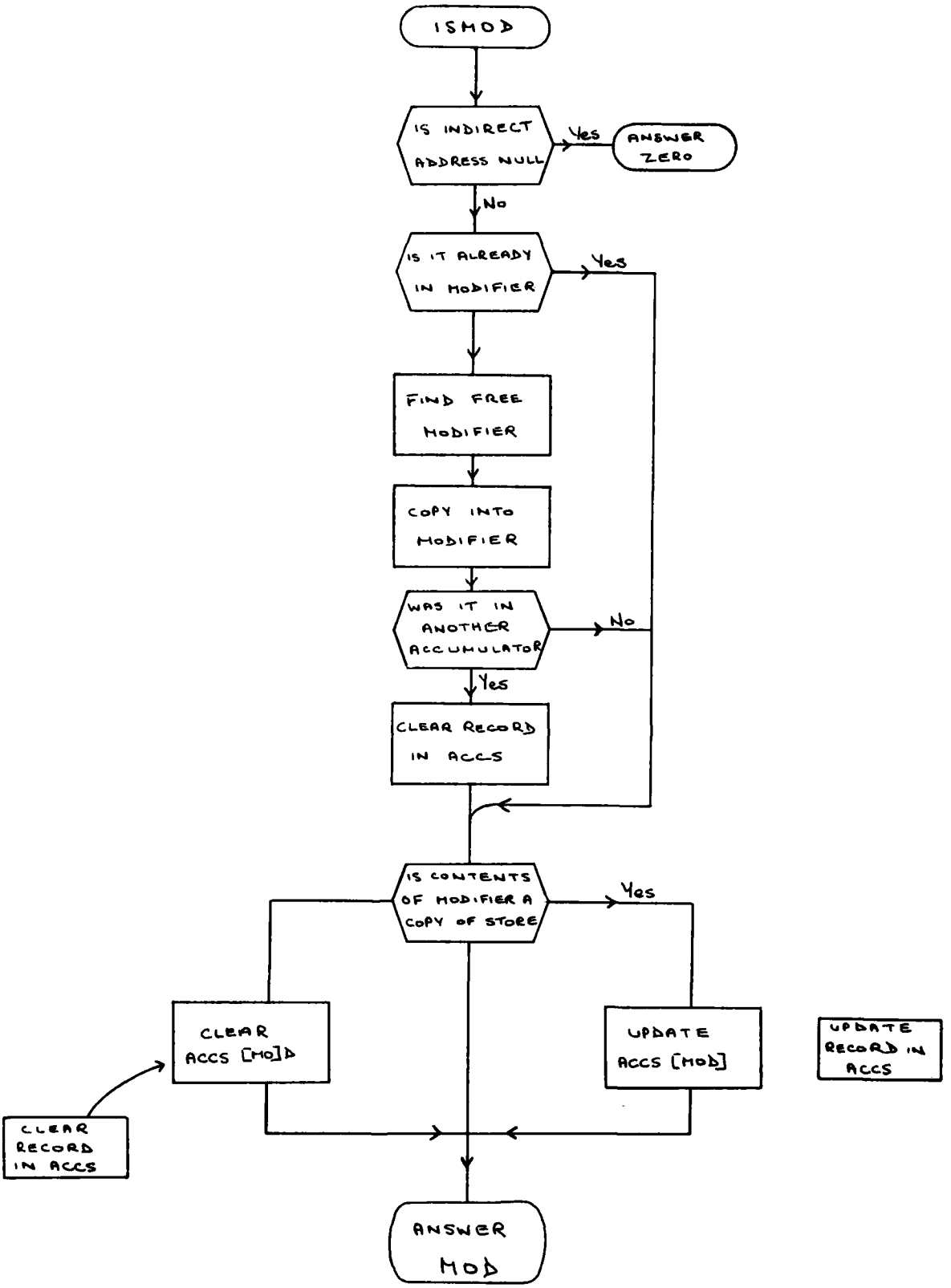
('VALUE' 'INTEGER' REF);

The result of this procedure is the number of the modifier holding the indirect address of the Arithmetic Operand referred to by REF. If the indirect address is null, the result is zero. If the indirect address is a modifier number, or if there is a copy of the required location already in a modifier, the result is this modifier number. Otherwise a call is made on FINDACC to obtain a modifier, and an instruction output to load it. Where the modifier contains a copy of a store location then the appropriate entry in ACCS is updated, otherwise it is cleared. ISMOD is called from INST, (the most common case) from SUBTERM in evaluating subscript expressions, and from STOREAWAY in the case of assignment and 'FOR' statements. A flowchart is given for this procedure.

'PROCEDURE' JOINCHAINS

('VALUE' 'INTEGER' CHAIN;
'LOCATION' 'INTEGER' MASTER);

This procedure joins the two jump chains, the chain CHAIN being joined on to the end of the chain MASTER. If the chain CHAIN is null, the procedure has no effect, and if the chain MASTER is null it is assigned the chain CHAIN. Otherwise a two-part directive is output to the loader, instructing it to join the chains. If however the chain MASTER is marked (-ve) this represents a label which has been set, and the chain CHAIN is set to this address.


ISMOD
IS INDIRECT ADDRESS NULL
Yes
ANSWER ZERO
No
IS IT ALREADY IN MODIFIER
Yes
FIND FREE MODIFIER
COPY INTO MODIFIER
WAS IT IN ANOTHER ACCUMULATOR
No
Yes
CLEAR RECORD IN ACCS
IS CONTENTS OF MODIFIER A COPY OF STORE
Yes
CLEAR ACCS [MOD]
UPDATE ACCS [MOD]
UPDATE RECORD IN ACCS
CLEAR RECORD IN ACCS
ANSWER MOD

'PROCEDURE' KILLEXPR;

This procedure is called at the end of nested expressions in cases where the result is no longer required. It is called directly from the syntax at the end of subscript expressions, from RELATION at the end of expression in conditions, and from DOSHIFT in the special case of expressions following shift operators. The stacked values of EXPRCHAIN,PREFFACC,EXPSCALE,SCALEFIRM, and BITSHIFT are restored by means of a call of OFFSTACK, and the pointers LH AND RH reset by means of a call of SETLHRH.

'INTEGER' 'PROCEDURE' LHACC;

This procedure is used to find whether the Arithmetic Operand next to the top of the stack (ie the LH operand) is held in an accumulator. If it is, then the result is the number of the accumulator, otherwise zero.

'INTEGER' 'PROCEDURE' LHEQRH;

This procedure is called after the first Term of the right hand side of an assignment statement. If the Term is identical to the variable on the left hand side, and it is neither floating point nor part-word, then the result is 1, otherwise the result is zero. This is used to detect cases such as: a←a+b.

'PROCEDURE' LIBTRACE

('VALUE' 'INTEGER' LNO,ADD,SPIEL);

This procedure is used to set up calls on trace procedures having one parameter, usually a string. The address of the string (ie run time address) is passed as the parameter ADD, LNO being the procedure reference number, and SPIEL the commentary. An instruction to load the address into accumulator 7 is output, followed by a call of CALLLIB to output the call of the procedure.

'PROCEDURE' LOCACT;

This procedure acts upon the Arithmetic Operand on the top of the stack. It alters the record, so that when used, the location of the operand rather than its value is obtained. LOCACT is called from the syntax in the case of the use of 'LOCATION'( ), and also from ANYMORE in the case of 'LOCATION' parameters.

If the direct address is non-zero then the type is changed to integer, and the marker bit LCM set. Otherwise the indirect address is copied into the direct address, the indirect address set to zero, and the type changed to integer, but without the location marker set. This second case optimises occurrences of anonymous references in location parameter positions.

'PROCEDURE' LOOKUP;

This is the procedure for looking up the records of non-label identifiers in a non-declaration context. This procedure forms an Arithmetic Operand on top of the stack and copies into it the four full word fields: TYPEBITS, PARTWORD/PARAMSPEC,DIRADD and INDADD. The commentary pointer SPIEL is set to the address of the identifier string in the stored record. If the identifier is used, but undeclared, a warning message is output, and the Arithmetic Operand set to type Integer, with a SPIEL of "(UNDECLARED)".

'INTEGER' 'PROCEDURE' LOOKUPLAB;

This procedure is used to scan the label list of the current block to find whether there is a label record with the same identifier string as that held in NAME. If there is not, then a new record is formed and added to the list by means of ONLAB. Thus the result of LOOKUPLAB is always a pointer to a label record.

'INTEGER' 'PROCEDURE' LOOKUPNAME

('VALUE' 'INTEGER' LIMIT,REF);

This procedure is used to search down the main declaration list, starting with the value held in DECLIST, and stopping when the value of the chain equals that of LIMIT. The identifier string of each record is compared with that of the record referred to by REF. If a match is found, then the result is a pointer to that record, otherwise zero.

'PROCEDURE' MAKEPARAM

('VALUE' 'INTEGER' TYPE);

This procedure is used to create the "hidden" type/scale parameters for untyped parameters and 'VALUE' procedures, and also for the result of intermediate calls of a multiple call of a typed procedure.

An Arithmetic Operand is created on the stack by means of a call of GRABVAR. Its TYPEBITS are set to 'INTEGER' and the accumulator number field inserted. This is obtained from the parameter specification record, the relevant entry of which is pointed at by PSP, which is incremented. The commentary is set to "(TYPE)".

If the parameter TYPE is zero, the record being set up is that of the result of an intermediate call of a multiple call. In this case the direct address of the Arithmetic Operand, and the appropriate entry in ACCS are updated by means of a call of TOPACC. Otherwise the typebits are marked to denote that the Arithmetic Operand is to be treated as a constant, and the direct address field set to the type/scale. If the parameter TYPE is two, this occuring for untyped parameters, the type is obtained from EXPSCALE. Otherwise (TYPE=1) the type/scale is obtained from the type of the procedure being called, which will have been changed to that of the context.

'PROCEDURE' MAKESPEC;

This procedure prepares a parameter specification record for a procedure identifier record. It is called from PROCSTACK in the case of a procedure declaration, and also from BEGINPSPEC in the case of a procedure specification.

The address of a seven word area of stack is set into PARAMPTR2, and this address plus the marker MARK set into the PARAMSPEC word of the procedure identifier record on the head of DECLIST. The marker indicates that the procedure has not yet been completely processed, and is used in the detection of recursion. The PAC field of IDTYPE2 is set to 7, and if the procedure is a 'VALUE' 'PROCEDURE' the fields SPB and PVL are also set for the context parameter. The values set up in PARAMPTR2 and IDTYPE2 will be placed in PARAMPTR and IDTYPE by a subsequent call of EXCPSPEC. This precaution is required in the case of a 'PROCEDURE' parameter of a procedure declaration.

```
'PROCEDURE' MASKINST

        ('VALUE' 'INTEGER' REF,ACC,SENSE);
```

This procedure is used to output mask instructions (function 15) for the packing and unpacking of data. The parameter REF is a reference to an Arithmetic Operand on the stack which is required to be packed or unpacked using the accumulator specified by ACC. The parameter SENSE is either all zeros, in which case the required field is unaltered, and the rest of the word cleared; or the parameter is all ones (-1), in which case the required field is cleared and the rest of the word unaltered.

```
'PROCEDURE' MOVE

        ('VALUE' 'INTEGER' N;
         'LOCATION' 'INTEGER' FROM,TO);
```

This procedure moves the number of words given by the parameter N, from successive locations starting from the location given by the parameter FROM, to successive locations starting from the location given by the parameter TO. The higher numbered locations being moved first. This is essential in the case where MOVE is called from ONLAB which is called by ENDLABBLOCK. In this case the areas given by TO and FROM may overlap, the data being moved upwards.

'PROCEDURE' NEWLINE;

This procedure outputs a newline (CrLf) using OUTCHAR

```
'INTEGER' 'PROCEDURE' NEXTCHAR

                ('LOCATION' 'INTEGER' Z);
```

This procedure unpacks one six-bit character, packed four characters to a word. The parameter Z specifies the location of a pointer to the packed characters. This pointer is incremented by one character position each time a character is unpacked. The two most significant bits of the pointer being used as an index to the character within the word. The bit pattern 00 specifies the character in the most significant six bits of the word.

'PROCEDURE' NEXTPARAM;

This procedure is used to build up the parameter specification record which is pointed at by the PARAMSPEC in a procedure identifier record. This record is a summary of the parameter requirements of a procedure, and is required to be available when a call of the procedure is to be compiled. This should be distinguished from the identifier records of the parameters themselves, which are only required whilst compiling the procedure; and are afterwards deleted.

NEXTPARAM is called directly from the syntax when dealing with procedure specifications, from NEWNAME when dealing with procedure declarations, and from BEGINPSPEC to set the (hidden) context parameter.

If, on entry, IDTYPE is marked negative (seventh parameter), then a call is made on XSPARAMS with a zero parameter. This causes a diagnostic message to be printed out using the current (procedure parameter) identifier. If IDTYPE is zero, there have been more than seven parameters, and no action is taken. Otherwise the current value of IDTYPE is added to the parameter specification record, the pointer PARAMPTR incremented, and the following word in the record set to MARK (end of record marker). The parameter accumulator field PAC of IDTYPE is decremented in the following sequence 7,6,5,3,2,1,0. If the value 0 is reached, IDTYPE is set to MARK to indicate that no further parameters are allowed. If the parameter is an untyped parameter (pair) then the field SPB of IDTYPE will be non zero. If it has the value 1, it is set to 2 and the bit LCM cleared. If it has the value 2, it is set to 1 and the bit LCM set to the value of the bit LM2. This bit will only be set in the case of 'LOCATION' parameters.

'INTEGER' 'PROCEDURE' NEXTPTYPE;

This procedure is a selector action, called directly from the syntax. It returns an integer in the range 0 to 3 depending on the type requirements of the next parameter. This is obtained by means of a call of PARAMCLASS, but after the values of SCALEFIRM and EXPSCALE have been set up.

If the parameter is untyped, these are cleared, otherwise EXPSCALE is set to the type of the parameter, and SCALEFIRM set to 1.

'PROCEDURE' NONAME;

This procedure forms an Arithmetic Operand on the top of the stack primarily for use with Anonymous References.

GRABVAR is used to set up the Arithmetic Operand, the commentary field SPIEL set to "(ANON)", and the TYPEBITS field set to Integer(Array). This procedure is called from the syntax and from other compiling actions to form an anonymous Arithmetic Operand.

```
'PROCEDURE' OCTLOOP

        ('VALUE' 'INTEGER' WORD,DIGITS);
```

This procedure is the basic octal print routine, it outputs the specified number of octal digits from the least significant end of the value supplied.

The parameter WORD is first shifted cyclically left so that the first octal digit to be printed is moved to occupy the most significant three bits. This is then repeatedly shifted three places left cyclically, and octal digits extracted from the least significant end. These are output using OUTCHAR. The number of digits to be output is specified by the parameter DIGITS.

```
'PROCEDURE' OFFSTACK

        ('VALUE' 'INTEGER' N;
         'LOCATION' 'INTEGER' START);
```

This procedure moves the number of words of data specified by N from the main stack to the area of core starting from START. Before the data is moved the variable whose location is passed as START points at the block of data on the stack. This procedure is essentially the inverse of ONSTACK and is used to "unwind" a level at the end of a syntactically recursive situation. After the data has been moved, the value of the variable STACKPOINTER is updated, as the area on the stack occupied by this data (and any data held above it) has now been relinquished.

```
'PROCEDURE' ONLAB

        ('VALUE' 'INTEGER' FROM);
```

This procedure moves a label record from the part of the stack whose location is given by the value of FROM, on to the end of the label stack, chaining it up to the chain LABDECLIST. This is used to move the record of a label on to the label stack, on its first occurrence within a block, and also during label stack compaction at the end of a block. It should be noted that the label stack is inverted, ie the "top" record on the stack occupies the lowest location.

The size of the record is first calculated, and then the new value to be assigned to the label stack pointer. The record is then moved. If LABDECLIST is zero, it is set to point to the record, otherwise the record below is set to point at the top record. Finally the label stack pointer LABSTACKPTR is set to its new value.

```
'PROCEDURE' ONSTACK

          ('VALUE' 'INTEGER' N;
           'LOCATION' 'INTEGER' START);
```

This procedure moves the number of words of data specified by N, from the area of core starting from START, on to the compiler's main stack. The variable location is passed as START is updated to point to the word on stack containing its previous contents. This procedure is used to stack the compiler's workspace in syntactically recursive situations, eg nested 'FOR' statements. It is also used to move declarations of new identifiers on to the stack and chain them up.

```
'PROCEDURE' OPERATE

          ('VALUE' 'INTEGER' FUN);
```

This procedure is used to apply logical and shift operators between the two Arithmetic Operands on top of the stack. No rescaling takes place prior to the specified operation, and the scale of the result is set as 'INTEGER'.

After ensuring that the left hand operand is in an accumulator, and that the right hand operand is unpacked if required, the instruction with the function code FUN is output using INST. The top operand is removed from the stack, and the type of the remaining operand changed to integer.

```
'PROCEDURE' OUT1014

          ('LOCATION' 'INTEGER' TA;
           'VALUE' 'INTEGER' TEN,ADD);
```

This procedure forms a 24 bit word from the ten bit group TEN and the least significant fourteen bits of the address ADD. This is then output using OUT24 with the transfer address given by TA and the address tag of ADD. This procedure is used to output instructions (Program), and constants containing address information (Special).

```
'PROCEDURE' OUT1014CS

          ('VALUE' 'INTEGER' ADD);
```

This procedure is used to output constants containing address information destined for the special constants (Special) area. The ten bit group to be packed in the most significant end is obtained from the variable TOPTEN.

```
'PROCEDURE' OUT24

          ('LOCATION' 'INTEGER' TA;
           'VALUE' 'INTEGER' WORD.TAG);
```

This procedure is the basic output procedure for 24 bit words which are to be relocated and loaded to core by the loader. The parameter TA specifies which transfer address is to be used to load the word (Program,Data or Special), and the address tag TAG specifies how the word is to be modified. The relocatable binary tag is formed from the address tag of the transfer address and the address tag from the parameter TAG. The appropriate transfer address is incremented by one.

('VALUE' 'INTEGER' CONST);

This procedure outputs whole word constants destined for the special constants (Special) area. These constants do not require relocation.

'PROCEDURE' OUT27

('VALUE' 'INTEGER' ADD,SPIEL);

This procedure is used to output instructions with a function part of OCTAL(27), an accumulator part of 4 and the address and commentary specified by ADD and SPIEL. This procedure is designed to be used to set up procedure calls, the link being set in accumulator 4. This procedure is also used for procedure exit,jumps to label parameters, and jumps to 'EXTERNAL', 'ABSOLUTE' and 'COMMON' labels. In these cases there is no need to set a link, but it is simpler to do so.

'PROCEDURE' OUT5

('VALUE' 'INTEGER' TAG,WORD);

This is the basic output procedure for relocatable binary. It outputs a five character group. The first six bit character is given as the parameter TAG, and the following four characters are unpacked from the parameter WORD. Before the first character is outout OUTDEV is changed to OCTAL(107) to select the punch, and after the last character is output OUTDEV is reset to OCTAL(106). A checksum TSUM is formed of all characters output. If the variable TEST is non-zero then the procedure does nothing.

'PROCEDURE' OUTCHAR

('VALUE' 'INTEGER' CHAR);

This procedure converts the character whose internal representation is given by the parameter CHAR to the appropriate ISO7 character. This is then output to the peripheral whose address is given by the variable OUTDEV. The internal representation for newline is 64, this being output as Carriage Return followed by Line Feed.

'PROCEDURE' OUTCJ;

This procedure is used to output all conditional jump instructions associated with conditions, except those of the form 'THEN' 'GOTO' Label, these being output by GOTOL. OUTCJ is called directly by the syntax to output a jump (if false) following the symbol 'AND'. It is called from ORACT, after the jump has been reversed, following the symbol 'OR' (jump if true). It is also called, following the symbol 'THEN' from STATUSTEST and from THENEX.

This procedure uses the previously calculated values of ACCUMULATOR and FUNCTION, the address part being obtained from SKIPCHAIN, which is updated.

'PROCEDURE' OUTCONT

('VALUE' 'INTEGER' WORD);

This procedure outputs the value passed as the parameter WORD as a five character relocatable binary group. The tag in this case is zero, signifying to the loader that this group forms part of a multi-group element. OUTCONT is often used as a parameter for DOSTRING.

'PROCEDURE' OUTI

('VALUE' 'INTEGER' XFM,N,SPIEL);

This is the basic procedure for the output of instructions (Program). The accumulator, function, and modifier field are passed as a packed ten bit field XFM. The address is passed as the parameter N, and a string for commentary as SPIEL.

Where the address is the address of an accumulator, it is converted to absolute form (X+4096) and a string formed of the form "@x" which replaces the string passed as SPIEL. Otherwise if the address is flagged as being the address of temporary workspace the string passed as SPIEL is replaced by "(TEMP)".

The instruction is output using OUT1014, and if the value of the variable LEVEL indicates that a detailed commentary is required, then this is output using DOSTRING to output SPIEL.

'PROCEDURE' OUTPRESETD

('VALUE' 'INTEGER' N,T);

This procedure is used to output preset data in data declarations. It is called from OUTPRESET to output numeric data, and also from PRESETSTRING to output the addresses of strings. (eg In the case of: 'INTEGER' S←"STRING";)

A check is made that presetting is allowed at this point, and also that there have not been more presets than data space allocated by the declaration. The preset value N is output by means of a call of OUT24, and relocated according to the address T. (Relocation is only required in the case of strings, which are stored in Special Data)

'PROCEDURE' OUTUJ;

This procedure is used to output unconditional jumps following 'ELSE', and also over procedures where required. The address part for the jump is obtained from SKIPCHAIN, which is updated.

'INTEGER' 'PROCEDURE' OUTWCONST

('VALUE' 'INTEGER' CONST);

This procedure outputs the constant CONST and returns the run-time address of the constant. As the loader optimises and allocates storage for the constants the address returned is always the same (W_0). The procedure outputs the constant as a prefix-continuation to the following instruction by the use of OUTCONT. The use of OUTWCONST as a separate procedure allows other methods of dealing with constants to be used if desired.

'PROCEDURE' OUTXFMN

('VALUE' 'INTEGER' X,F,M,N,SPIEL);

This procedure is used to output instructions (Program) where the fields of the instruction have been obtained separately. It packs the accumulator (X), function (F), and modifier (M) fields as a ten bit group. This together with N and SPIEL are passed to OUTI for subsequent output.

'INTEGER' 'PROCEDURE' PARAMCLASS;

This procedure returns an integer, in the range 0 to 3, depending upon the class of the parameter currently being pointed at, in a parameter specification record, by PSP. The classification is as follows:

Result	Parameter Type
0	'VALUE' (type)
1	'LOCATION' (type)
2	'ARRAY','SWITCH', and 'PROCEDURE'
3	'LABEL'

'PROCEDURE' PERM;

This procedure exchanges the top two Arithmetic Operands on the stack (ie the LH and RH operands), and updates the accumulator record, ACCS, by means of calls on ACCUPDATE.

'PROCEDURE' PICK

('VALUE' 'INTEGER' REF,ACC,SCALE,NEG);

This is the procedure responsible for the output of instructions for copying the Arithmetic Operand referred to by REF, into the accumulator ACC. It is unpacked, if required, rescaled to the type/scale SCALE, and negated if NEG is non zero. Conversions from 'FIXED' to 'INTEGER' are rounded.

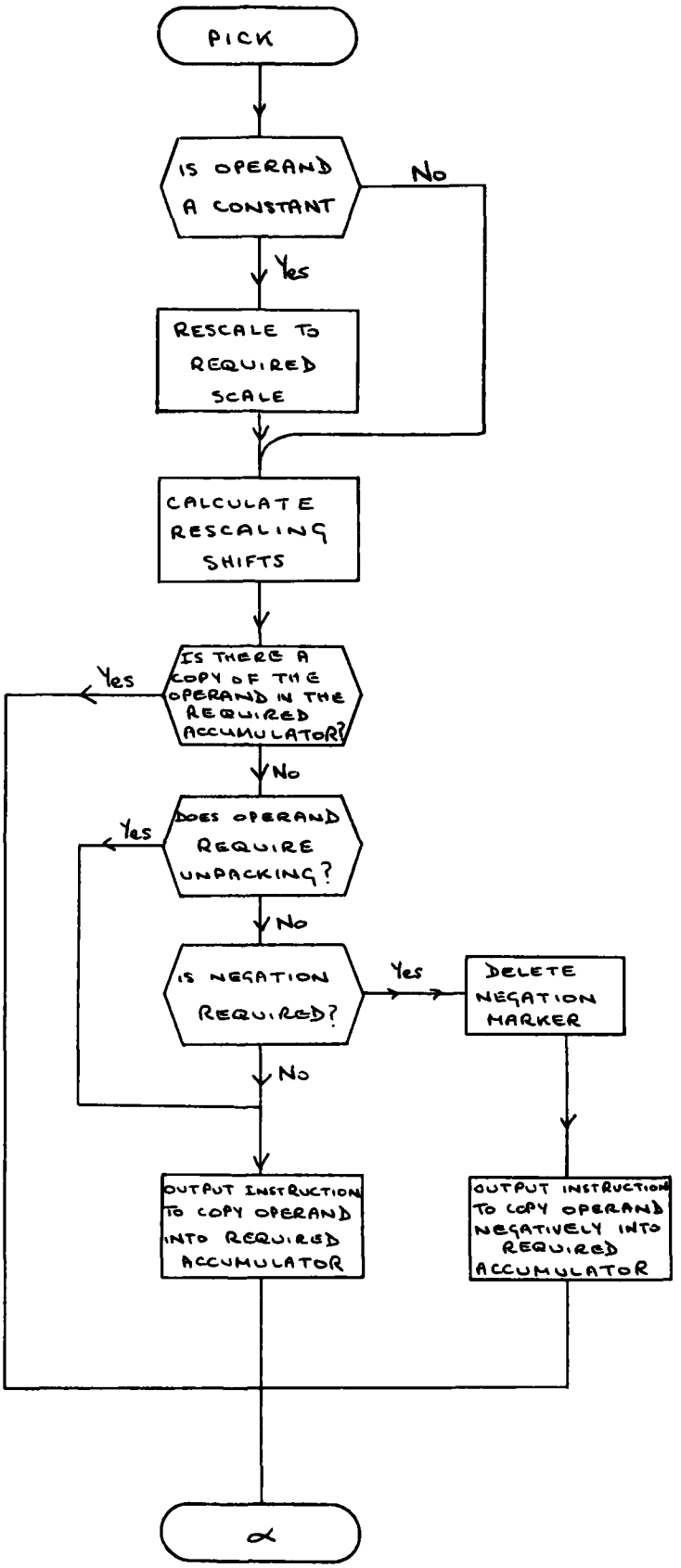
This procedure is not used with 'FLOATING' operands.

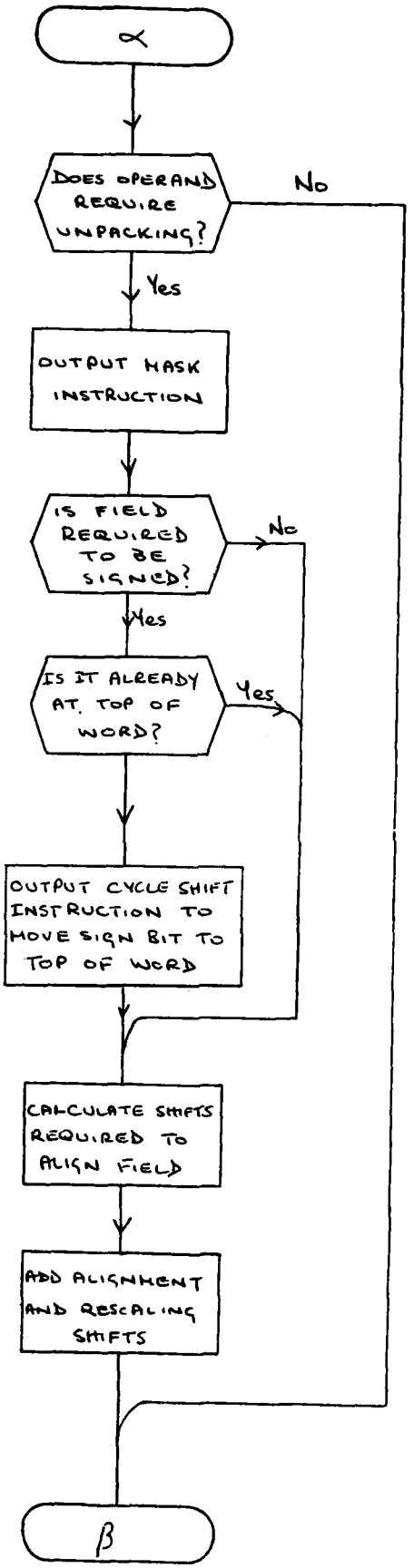
Detailed flowcharts describing the operation of the procedure are given.

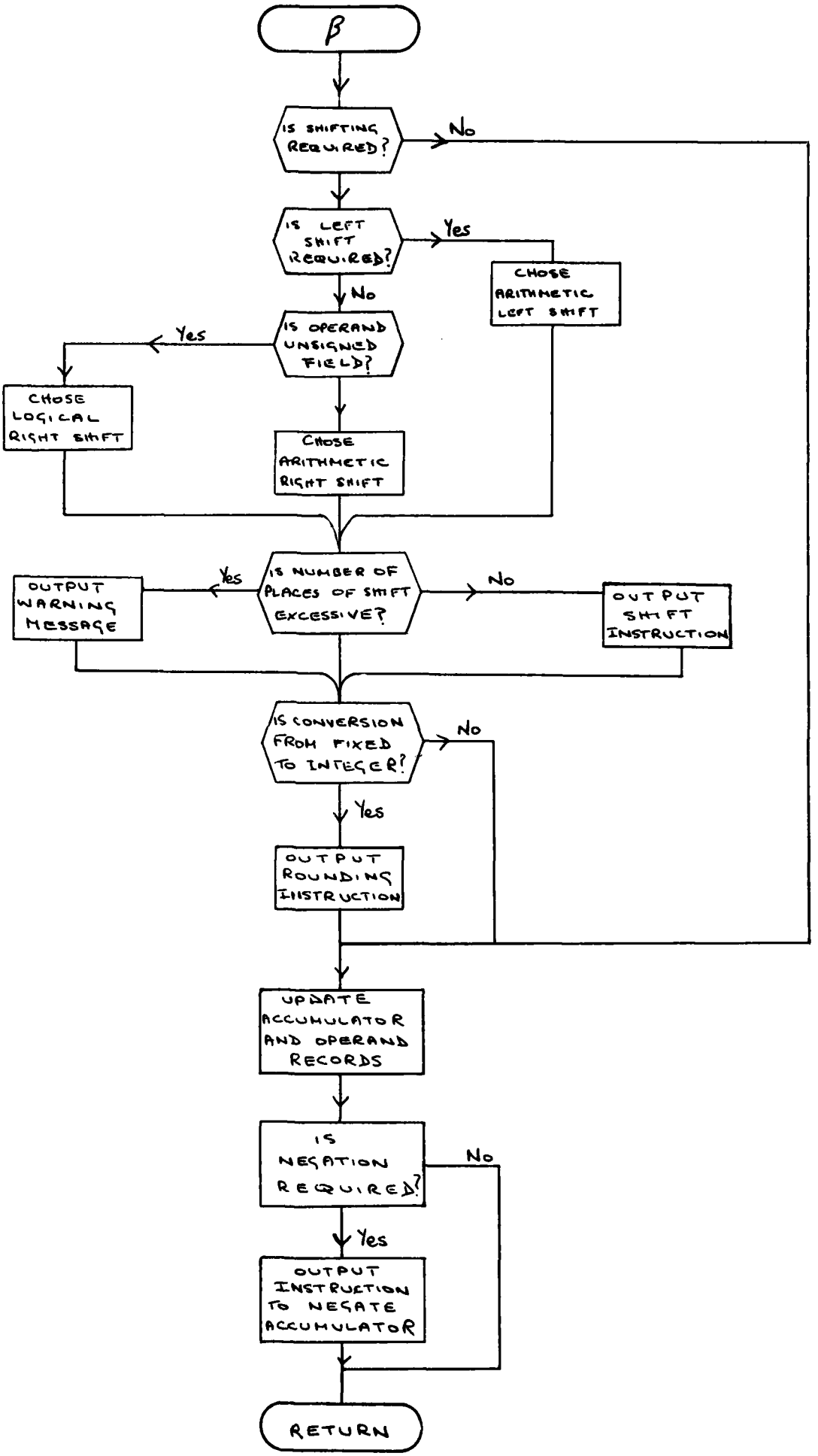
'INTEGER' 'PROCEDURE' POWERTWO;

If the top operand on the stack is a positive integer constant which is a positive power of two, then the result of this procedure is the power of two, otherwise the result is zero.

This procedure takes advantage of the fact that such a constant, 2^n, requires n+2 significant bits to be represented as a signed integer. This value is obtained from SGB. The cyclic shift used is equivalent to a division by 2^n, but with the remainder placed in the top of the word. Only if the constant is positive, and a power of two, will the result be one.


PICK
IS OPERAND A CONSTANT
No
Yes
RESCALE TO REQUIRED SCALE
CALCULATE RESCALING SHIFTS
IS THERE A COPY OF THE OPERAND IN THE REQUIRED ACCUMULATOR?
Yes
No
DOES OPERAND REQUIRE UNPACKING?
Yes
No
IS NEGATION REQUIRED?
Yes
DELETE NEGATION MARKER
No
OUTPUT INSTRUCTION TO COPY OPERAND INTO REQUIRED ACCUMULATOR
OUTPUT INSTRUCTION TO COPY OPERAND NEGATIVELY INTO REQUIRED ACCUMULATOR
α


α
DOES OPERAND REQUIRE UNPACKING?
No
Yes
OUTPUT MASK INSTRUCTION
IS FIELD REQUIRED TO BE SIGNED?
No
Yes
IS IT ALREADY AT TOP OF WORD?
Yes
OUTPUT CYCLE SHIFT INSTRUCTION TO MOVE SIGN BIT TO TOP OF WORD
CALCULATE SHIFTS REQUIRED TO ALIGN FIELD
ADD ALIGNMENT AND RESCALING SHIFTS
β


β
IS SHIFTING REQUIRED?
No
IS LEFT SHIFT REQUIRED?
Yes
CHOSE ARITHMETIC LEFT SHIFT
No
IS OPERAND UNSIGNED FIELD?
Yes
CHOSE LOGICAL RIGHT SHIFT
CHOSE ARITHMETIC RIGHT SHIFT
IS NUMBER OF PLACES OF SHIFT EXCESSIVE?
Yes
OUTPUT WARNING MESSAGE
No
OUTPUT SHIFT INSTRUCTION
IS CONVERSION FROM FIXED TO INTEGER?
No
Yes
OUTPUT ROUNDING INSTRUCTION
UPDATE ACCUMULATOR AND OPERAND RECORDS
IS NEGATION REQUIRED?
No
Yes
OUTPUT INSTRUCTION TO NEGATE ACCUMULATOR
RETURN

'PROCEDURE' PRINTADD

('VALUE' 'INTEGER' ADD);

This procedure prints the address given as the parameter ADD in the form of a single letter representing the tag type, and five octal digits representing the index. This is followed by the characters ": ".

'PROCEDURE' PRINTBUFF;

This procedure is used, after the detection of an error, to print the contents of the cyclic buffer INBUFF. This contains the last 64 non ignored characters read from the input tape. If less than 64 characters have been read, then only those will be output. The output consists of the characters "? . . ", the contents of the buffer, and the characters"←←←". Where the buffer contains a newline, this is followed by the character "?". This is therefore the first character on any line output by this procedure.

'PROCEDURE' PROCSTACK;

This procedure is called to perform the housekeeping required at the start of a procedure declaration. It is called from BEGINPROC for procedures within a segment, and also from SETLIBSEG for Library Procedure segments. STATUS is set to 1, as the procedure is entered by a jump, and a parameter specification record prepared by means of a call of MAKEPSPEC. A call is made on ONSTACK to move the current values of PROCCHAIN,PROCPTR,PARAMPTR, PARAMDECS,LINK,PROCSTRING,EXITCH, and DUECHAIN. This is only necessary where a procedure is declared within a procedure, but is carried out in all cases for the sake of simplicity. BEGINBLOCK is then called to set up an outer block for the procedure which encloses the parameters (if any). PROCPTR is set to point to the procedure identifier record, and LOCALLIMIT moved one record beyond this. This, and the call of BEGINBLOCK, prevent the redeclaration of the procedure and parameter identifiers within the outer block of the procedure, thus rendering the parameters inaccessible. EXITCH and PROCSTRING are set to zero to inhibit procedure trace unless specifically required.

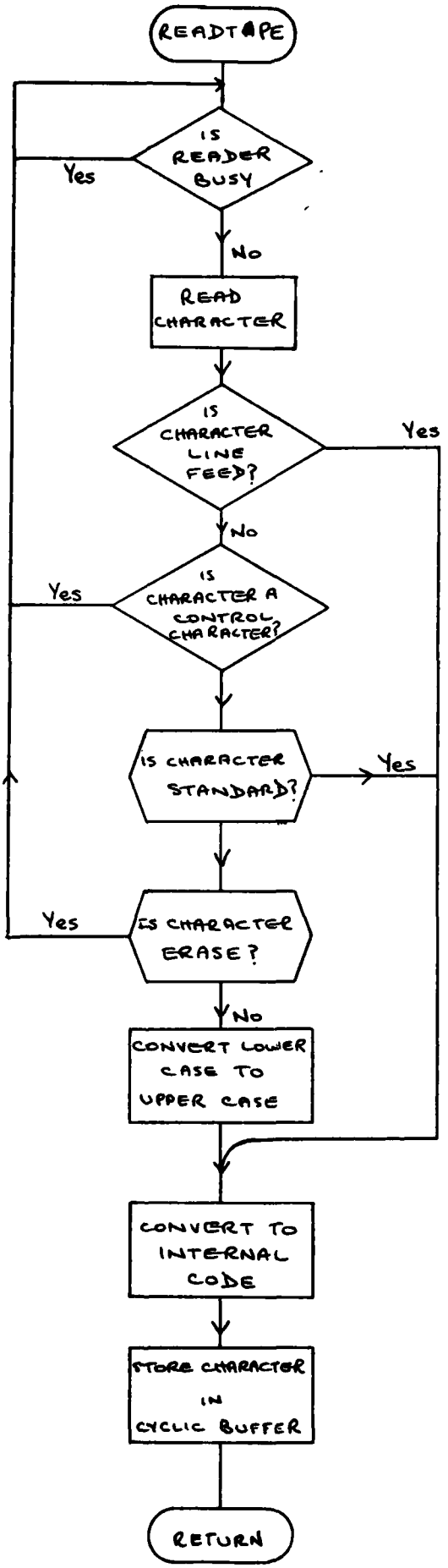
'PROCEDURE' PUNCHLOOP

('VALUE' 'INTEGER' CHAR,TIMES);

This procedure outputs the binary character CHAR directly to the punch, the number of repetitions being given by the parameter TIMES. This is used only to obtain blank tape and groups of erases. No characters are output if the variable TEST is non zero.

'INTEGER' 'PROCEDURE' READTAPE;

This procedure is the basic input procedure of the compiler. It reads one character at a time from paper tape and converts it to internal representation. Line feed is treated as newline (64), all other control characters and erase being ignored. Lower case letters are mapped onto the corresponding upper case letters. After conversion the character is also deposited into a 64 character cyclic buffer INBUFF, using an integer INCTR as the pointer to the current character position. This pointer always has a value in the range 0 to 63 inclusive. A flowchart is given for this procedure.


READTAPE
IS READER BUSY
Yes
No
READ CHARACTER
IS CHARACTER LINE FEED?
Yes
No
IS CHARACTER A CONTROL CHARACTER?
Yes
IS CHARACTER STANDARD?
Yes
IS CHARACTER ERASE?
Yes
No
CONVERT LOWER CASE TO UPPER CASE
CONVERT TO INTERNAL CODE
STORE CHARACTER IN CYCLIC BUFFER
RETURN

'INTEGER' 'PROCEDURE' RESCALE

('VALUE' 'INTEGER' NUMBER,OLDSCALE,NEWSCALE);

This procedure rescales the constant NUMBER whose type/scale is given by OLDSCALE to the type/scale required by NEWSCALE. Type changing from 'INTEGER' to 'FIXED' or 'FLOATING', or from 'FIXED' to 'FLOATING' is allowed, other type changes are not. This means that constants with a decimal point cannot be used in a strictly 'INTEGER' context, eg as a subscript. The number type is given by 'BITS' [3,10] and the scale by 'BITS' [6,18] of OLDSCALE and NEWSCALE.

Provided that rescaling is required, the number is first normalised and its scale adjusted as required. If a 'FLOATING' result is required then the number is truncated, rounded, and packed. Otherwise the number is fixed to the required scale, provided this can be done without total loss of significance; and rounded. A flowchart is given for this procedure.

'PROCEDURE' REVCHAIN;

This procedure exchanges the chains DUECHAIN and SKIPCHAIN.

'PROCEDURE' REVCJ;

This procedure is used to reverse the type of conditional jump instruction calculated but not yet output. Function 20 is changed to 21, and function 22 to 23 and vice versa. In the case of function 24 the accumulator number is changed from 0 to 1, or from 1 to 0. This reversal of the test is required following the symbols 'THEN' 'ELSE','THEN' 'GOTO', and 'OR'.

'INTEGER' 'PROCEDURE' RHACC;

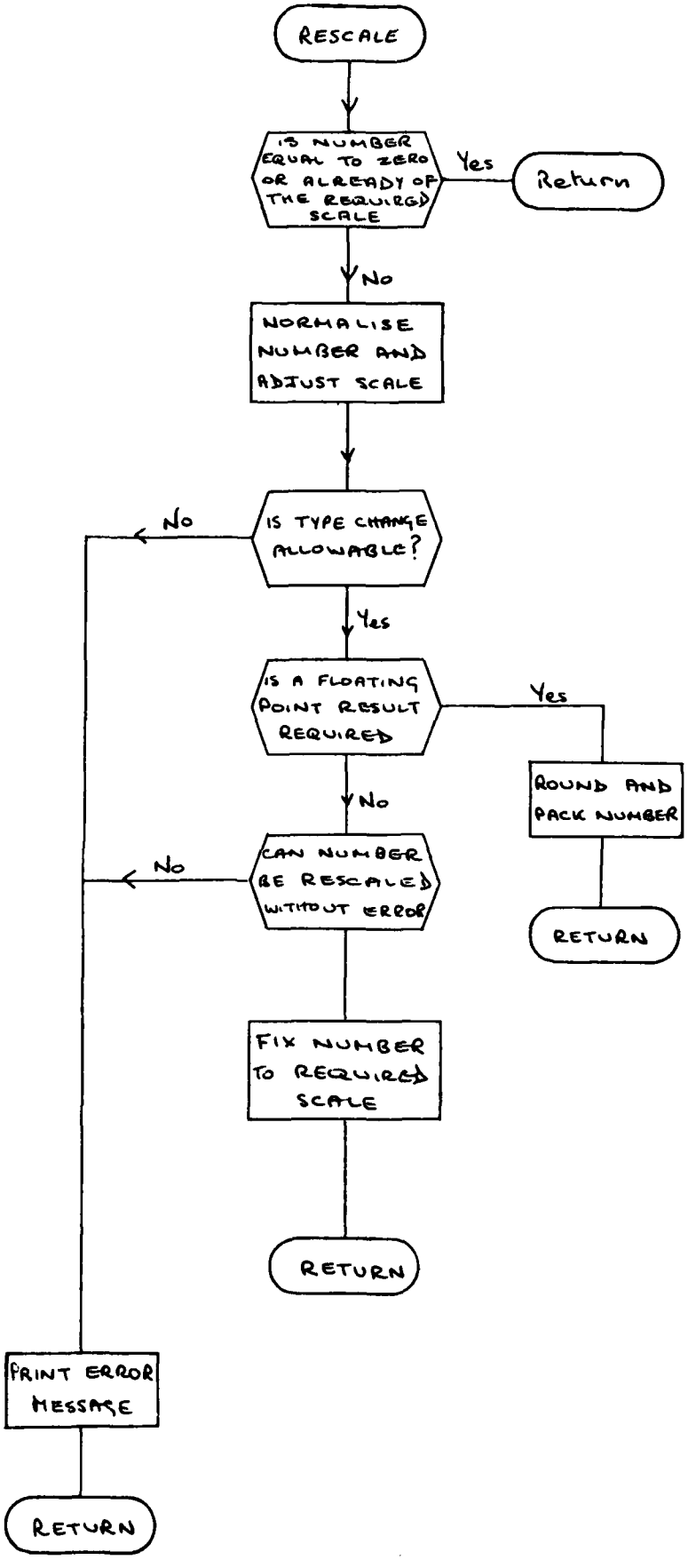
This procedure is used to find whether the Arithmetic Operand on the top of the stack (ie the RH operand) is held in an accumulator. If it is, then the result is the number of the accumulator, otherwise zero.

'INTEGER' 'PROCEDURE' RHSPTEST;

This procedure is a selector action called directly by the syntax, after the occurrence of an unsubscripted identifier in an expression. If the identifier is the name of a procedure, a result of 1 is returned to the syntax analyser, otherwise a result of 0 is returned. This allows the case of procedures without parameters to be detected, and differentiated from simple variables.

'INTEGER' 'PROCEDURE' SCALECON;

This procedure rescales the constant held in NUMBER, and whose scale is held in NUMBERSCALE, to the scale required by the current context which is held in IDTYPE. This procedure is called by the syntax to rescale Library Preset variables and scaled constants occuring in the operand field of code instructions. It is also called from other compiling actions which output preset values.


RESCALE
IS NUMBER EQUAL TO ZERO OR ALREADY OF THE REQUIRED SCALE
Yes
Return
No
NORMALISE NUMBER AND ADJUST SCALE
IS TYPE CHANGE ALLOWABLE?
No
Yes
IS A FLOATING POINT RESULT REQUIRED
Yes
ROUND AND PACK NUMBER
RETURN
No
CAN NUMBER BE RESCALED WITHOUT ERROR
No
FIX NUMBER TO REQUIRED SCALE
RETURN
PRINT ERROR MESSAGE
RETURN

'PROCEDURE' SCALENUM

('VALUE' 'INTEGER' REF,SCALE);

This procedure rescales the constant held in the Arithmetic Operand record referred to by REF, to the type/scale SCALE; changing the TYPEBITS of the record to the required scale. This is performed by means of a call of RESCALE.

'PROCEDURE' SCALETEST

('VALUE' 'INTEGER' A,B;
'SWITCH' S);

This procedure is supplied with two scale/type words A and B. It checks these for compatability, and jumps to the appropriate entry of the switch S according to the mode of the result of an operation with two operands of the specified types. If the four types considered are represented as follows:

Io = 'INTEGER' of unspecified significance
Is = 'INTEGER' of specified significance
Fx = 'FIXED'
Fl = 'FLOATING'

then the operation of the procedure may be described in the table below:

Case	Result	Entry of S
Io,Io	Io	1
Io,Is & Is,Io	Io	1
Is,Is	Is	2
Is,Fx (Io,Fx*)	Fx	3
Fx,Is (Fx,Io*)	Fx	4
Fx,Fx	Fx	5
Fl,Any	Fl	6
Any,Fl	Fl	6

Where the two scales are Io and Fx, in either order, a warning message is output as the result of the operation may be ill defined because the number of significant bits in the Integer Operand is not known, and hence its tolerance to rescaling cannot be predicted.

'INTEGER' 'PROCEDURE' SCANLAB

('VALUE' 'INTEGER' PTR);

This procedure scans the label list of the current block to discover whether there is a label record with the same identifier string as that of the record on the main (declaration) stack pointed at by PTR. If there is, then the result is the pointer to the label record, otherwise zero. This procedure is called by LOOKUPLAB to deal with the normal use of labels within a segment. It is also called by ENDPROG to check whether labels unset within a segment are 'COMMON' labels.

'INTEGER' 'PROCEDURE' SELOPTA;

This procedure is a selector action called by the syntax, after the first, but before the second, term on the right hand side of an assignment statement.

If the assignment function, ASSFUN, is marked negative, this marker is deleted, and if the resulting value is OCTAL(10) a call is made on UNARYMINUS. This case occurs where the right hand side commences with an unary minus sign, and the first term is not identical to the left hand variable.

If, now, the value of ASSFUN is OCTAL(10), then assignment optimisation is not required, and a result of zero returned to the syntax analyser. This causes it to chose the rule for the continuation of the expression in the normal manner. If however assignment optimisation is invoked, the top Arithmetic Operand record is deleted from the stack, and the result 1 returned to the syntax analyser. This causes it to chose the rule which effectively restarts the expression at the second Term, to evaluate from this Term onwards, ignoring the first. The result of this expression will be added to, or reverse subtracted from, store.

'INTEGER' 'PROCEDURE' SETASS;

This procedure is a selector action called by the syntax after the Becomes symbol in an assignment statement.

The value of ASSFUN is set to its initial value of OCTAL(10), the required scale for the right hand side, EXPSCALE, obtained from the scale of the left hand side, and SCALEFIRM set to 1, to indicate that this scale is mandatory. If assignment trace is required then PREFACC is set to 7, as the trace procedure requires it in this accumulator. The result 1 is returned to the syntax analyser. This causes the rule for expression which does not have assignment optimisation actions embedded to be selected.

If no trace is required then PREFACC is set to a value supplied by FINDACC, and the result zero returned to the syntax analyser. This causes the rule for expression which includes assignment optimisation actions to be selected.

'PROCEDURE' SETCHAINTOPTA

('VALUE' 'INTEGER' CHAIN);

This procedure sets the jump chain CHAIN to the current program address. As this represents a point which will be jumped to in the object program, the array ACCS is cleared. The chain is set by outputting the appropriate directive tag (17) to the loader. If the chain is null the procedure has no effect.

'PROCEDURE' SETDUE;

This procedure is used to set the chain DUECHAIN to the current program address. After the chain has been set DUECHAIN is set to zero.

'PROCEDURE' SETLABEL

('VALUE' 'INTEGER' LAB);

This procedure is used when the label specified by the pointer LAB is to be set to the current program address. This is done by means of a call of FIXLABEL. If label tracing is required a call is set up on L3 with the label identifier string as a parameter, by means of a call of TRACESTRING.

SETLABEL is called from SETLAB to set labels explicitly set in a segment, from ENDPROG to set labels which are supplied by a communicator, and from PROCENTRY to set labels within a procedure which are supplied as 'LABEL' parameters.

'PROCEDURE' SETLHRH;

This procedure sets the pointers LH (left hand) and RH (right hand) to the two top Arithmetic Operands on the stack. This procedure is used frequently, to ensure that these pointers are always up to date whilst compiling expressions.

'PROCEDURE' SETRT;

This procedure is used in the compilation of For statements. It is called when it is discovered that the For statement is of such complexity that the controlled statement is required to be treated as an anonymous procedure. This condition occurs where the for-list is more complex than the form.

expression comma expression while.

If the first element of the for-list is a simple expression, then the assignment of this value is followed by an unconditional jump whose address is recorded in RTA. If on entry to the procedure, RTA is non zero, a special directive (35) is output to the loader requesting it to change the unconditional jump into a jump-setting-link instruction. A location is allocated in the Special Data area, its address being recorded in RTA, and the location initially set to zero. This will later be set to the entry address for the controlled statement.

'INTEGER' 'PROCEDURE' SETUPPROC;

This procedure is a selector action called directly from the syntax, after the occurrence of an identifier which has been found to be the name of a procedure. If the procedure requires parameters, a result of 1 is returned, otherwise 0.

Before returning to the syntax analyser, the opportunity is taken to perform the housekeeping required to prepare for the generation of a procedure call.

If the call is recursive an error message is output. The values of EXPRCHAIN,PREFACC,EXPSCALE,SCALEFIRM,BITSHIFT,PNP,PSP, and FPP are placed on the stack by means of a call of ONSTACK. This protects the workspace of any enclosing expression or procedure call. PNP (procedure name pointer) is set to point at the Arithmetic Operand record of the procedure to be called, FPP (first parameter pointer) set to point to the place on the stack where

'INTEGER' 'PROCEDURE' SETUPPROC/

the record for the first parameter will be created, and PSP (parameter specification pointer) set to point (one back) at the first entry in the parameter specification record of the procedure. The type of the procedure held in the Arithmetic Operand record is then changed to the type of the result of the procedure. If the procedure is a 'VALUE' 'PROCEDURE' this type will depend on the context in which the call is made. This is determined from the variables of the enclosing expression context, which at this point have not yet been altered. If the scale is firm, then this scale is used, otherwise the type 'FLOATING' is used. A call is made on MAKEPARAM to create an Arithmetic Operand record for the context parameter.

'INTEGER' 'PROCEDURE' SPECSTRING;

This procedure outputs the string held in the base of the stack, starting in the location overlayed with the identifier NAME. This string is output destined for the special constants (Special) area. The result of the procedure is the value of the Special Transfer Address (STA) before the string had been output. This procedure is called from the syntax analyser in cases where a string occurs in a 'SPECIAL' 'ARRAY'. In this case the result is irrelevant.

'PROCEDURE' STACKCHECK;

This procedure checks whether the main stack, which extends upwards, has collided with the label stack, which extends downwards. If a collision occurs the compilation is halted.

'PROCEDURE' STACKEXPR;

This procedure is called where it is required to nest expressions. It is called from the syntax in the cases of untyped bracketed expressions and expressions in conditions, from TYPEEXPR in the case of typed bracketed expressions, and from SETUPSUB in the case of subscript expressions. It should be noted that a bracketed expression following a shift operator is treated as a subscript expression, as the result is required to be of integer scale, and be in a modifier if evaluated.

The values of EXPRCHAIN,PREFACC,EXPSCALE,SCALEFIRM and BITSHIFT are stacked by means of a call of ONSTACK. EXPSCALE and SCALEFIRM are then cleared in case the expression to be evaluated is untyped.

'PROCEDURE' STARTSEG

('VALUE' INTEGER' TYPE);

This procedure is called at the start of Program (TYPE=1), Library (TYPE=2), and Common (TYPE=3) segments. It punches the segment header on tape, and prints a start of segment message. The transfer addresses PTA,DTA,STA, and the allocation variables DATASTART and DATAMAX set to their respective initial values. ZEROACCS is called to clear ACCS.

'PROCEDURE' STATUSCHECK;

This is a compiling action which is called directly from the syntax, and also from other compiling actions.

It is called at the start of every statement except 'CODE' and 'GOTO' statements. STATUSTEST is called, with DUEERR as a parameter, so that if STATUS is 1 or 2 and DUECHAIN is null, then an error is reported. If STATUS is 2,3, or 4 then the deferred action is carried out.

'PROCEDURE' STATUSCODE;

This is a compiling action which is called at the start of 'CODE' statements. It ensures that any deferred action associated with values of STATUS of 2,3, or 4 are carried out. This is performed by means of a call of STATUSTEST with DUMMY as a parameter, as a code statement need not be entered at the beginning. A call is also made on ZEROACCS to clear the accumulator record ACCS.

STATUSCODE is also called from SETLAB as the above actions are required at this point, but for different reasons.

'PROCEDURE' STATUSTEST

('PROCEDURE' PROC);

This procedure is called at the start of every statement except 'GOTO' statements. The value of the variable STATUS indicates the type of the preceding statement, as follows:

STATUS	Preceded By
4	'THEN' (or 'THEN' 'ELSE')
3	'ELSE'
2	'ANSWER' statement
1	'GOTO' statement (also certain cases of 'DO' statement, and also where the statement is the first statement of block containing the declaration of a procedure)
0	All other cases. (And also where the statement is labelled)

The use of this method ensures that the number of unconditional jumps generated is reduced as far as possible. In this case it is necessary to defer the actions associated with certain symbols until the following symbol(s) have been processed. A value of 2,3, or 4 for the variable STATUS indicates that an action has been deferred. A value of 0 indicates that control is passed normally to the statement, but a value of 1 indicates that control is not passed normally to the statement, and if in this case DUECHAIN is null then the statement is inaccessible. In this case a call is made on the procedure parameter PROC. STATUSTEST is called from STATUSCHECK with DUEERR as a parameter, and from STATUSCODE with DUMMY as a parameter. In the latter case it is assumed that a code statement which is not directly accessible, is entered via a relative jump.

'PROCEDURE' STOPOP

('VALUE' 'INTEGER' TA,TB);

This procedure is called in cases where a recoverable error, but of sufficient gravity to prevent the compiled program from running safely, is discovered. This procedure outputs the two strings TA and TB separated by " : ". If the variable TEST is zero then it is set to one and the message " : COMPILATION INHIBITED" output. The effect of setting TEST to a non zero value is to inhibit further output on the punch until the end of the segment.

This is followed by a newline, and PRINTBUFF is called to print the contents of the input buffer.

'PROCEDURE' STOREAWAY

('VALUE' 'INTEGER' TFLAG);

This procedure is the procedure that generates the instructions to assign an expression to a variable. It is called from STORE in the case of an assignment statement, and from FORINC and ASSCV in 'FOR' statements. The parameter TFLAG indicates whether assignment trace is required, and is zero in the second two cases. The variable ASSFUN indicates which function is to be used for the assignment. Where this is marked negative, the assignment is of zero, and is treated as a special case. Where ASSFUN is zero, no code is generated; this occurs in the case of dummy assignments, e.g. 'FOR' I←I 'WHILE' . .

Provided that ASSFUN is non zero, the procedure starts by detecting whether the assignment is of the form partword becomes constant, and that trace is not required. If this case is detected then FFLAG is set. Next an address ADD, is calculated. This will be inserted into the entry in ACCS of the accumulator used for the assignment. If the left hand variable is not partword and not indirectly addressed then this address is used, otherwise the address of the right hand side is used if possible. A suitable accumulator is then chosen, bearing in mind that the right hand side may already be in an accumulator. If trace is required then the accumulator must be 7, otherwise if the assignment is of the form wholeword becomes zero, then accumulator 0 is chosen.

If the indicator FFLAG is clear then the expression on the right hand side is copied into the chosen accumulator, unpacked, and rescaled as required. This is omitted if the accumulator is zero. Otherwise if the left hand side is indirectly addressed the entry in ACCS for the chosen accumulator is set to all ones to inhibit the possible choice of this accumulator as a modifier.

If trace is required then the instructions required are output. If the index of the left hand side is in a modifier, this is first dumped. A call on L7 is then output, with the result in accumulator 7, the type/scale in accumulator 6, and the address of the string, which is output to Special Data, in accumulator 5.

If the assignment is to a wholeword then the assignment instruction is then output; this will use one of the four functions STO,STN,ADS, or SSB. Otherwise a modifier is chosen, (this being zero if modification is not required) and the number of shifts to align the expression to the field in which it is to be stored, calculated. If FFLAG is clear then instructions are output to perform the assignment. The expression is first shifted, if required, then masked, and exchanged with the word of

'PROCEDURE' STOREAWAY/

store. This is then masked to clear the field to which the expression is being assigned, and the remaining fields replaced by adding them back to store. Otherwise if FFLAG is set, the constant is shifted and masked to fit the field. If the left hand side is not already in the accumulator then instructions are output to copy it into the chosen accumulator. Provided the assignment is not of all ones to the field, an instruction is then output to mask the field. Provided the assignment is not all zeros, then an instruction is output to or the constant in to the field. Finally the word is stored.

At the end of the procedure the record of any previous copies of the left hand side is deleted, the right hand Arithmetic Operand deleted from the stack, and the accumulator record for the accumulator ACC updated with the address ADD.

A simplified flowchart is given.

'PROCEDURE' SUBTERM

('VALUE' 'INTEGER' NEG);

This procedure is called after each Term of a subscript expression. From PLUSSUB with NEG equal to zero, and from MINUSSUB with NEG equal to one. This deals with each Term individually in order to reduce the amount of code generated. When the procedure is called, the pointer RH points at the Arithmetic Operand which is the Term of the subscript expression, and the local pointer LH is set to point at the Arithmetic Operand of the variable, (which may be anonymous) which is to be subscripted. If the Term is a constant then this is incorporated in the direct address of the variable. If the term is not a constant, but the indirect address of the variable is null, then this is set to the direct address of the term, which is evaluated in a modifier if required. Otherwise a suitable modifier is chosen, and the term added or subtracted from this modifier. A detailed flowchart is given.

'PROCEDURE' SUSPIEL;

This procedure is called after the expressions for the step and until elements of a For statement have been processed. This alters the commentary of the two top Arithmetic Operands on the stack to "(STEP)" and "(LIMIT)".

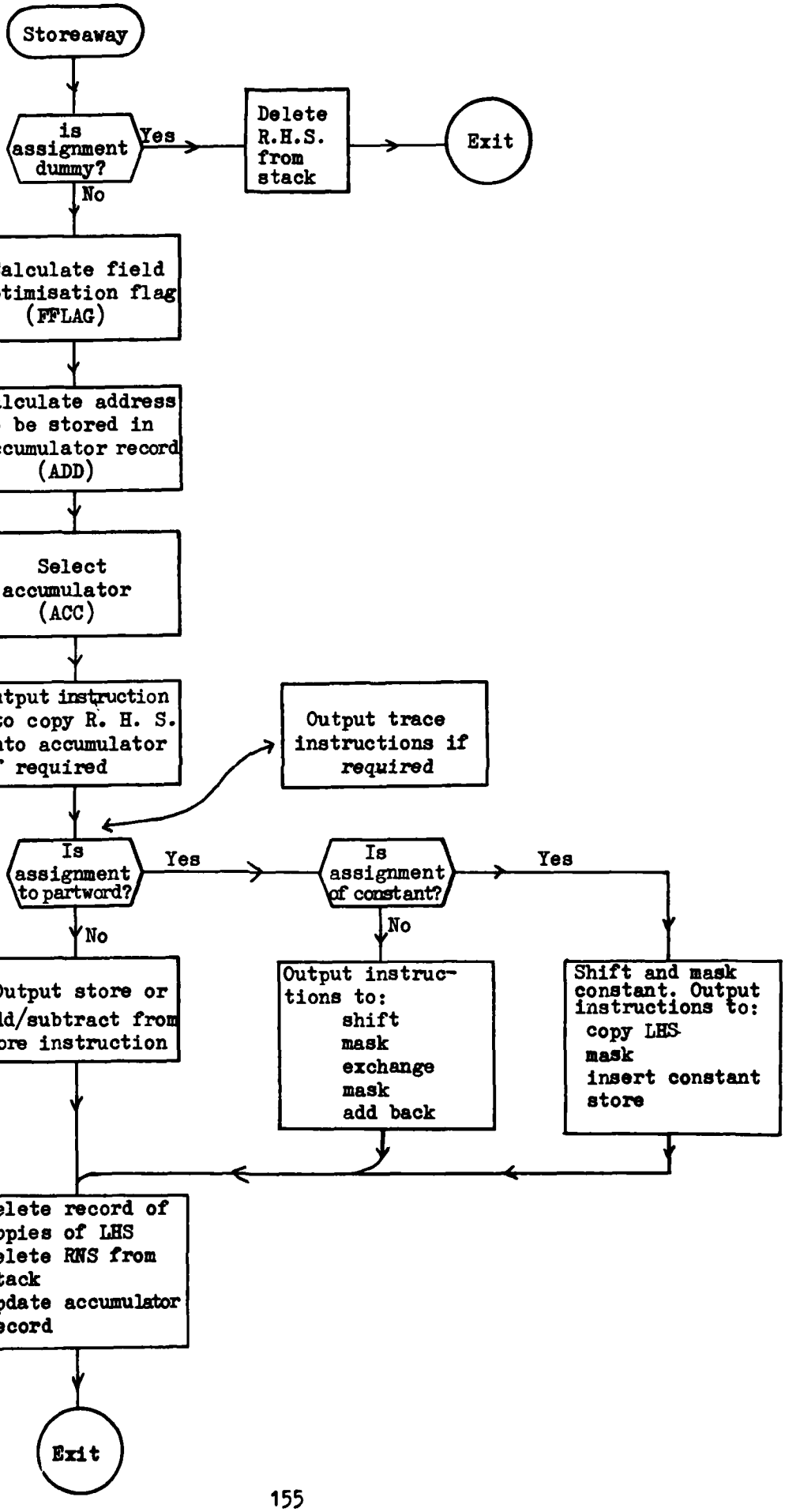
'PROCEDURE' SWOP

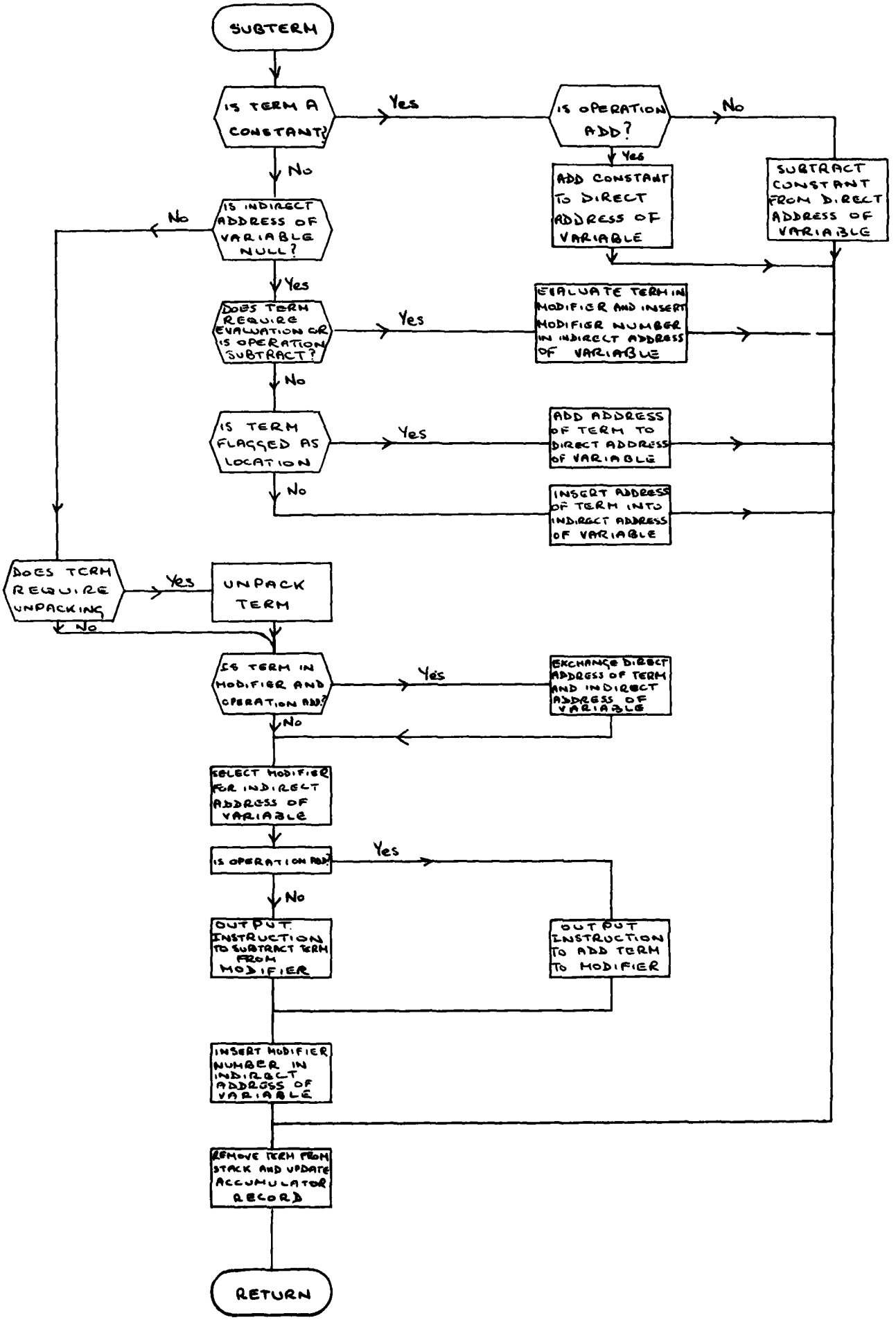
('LOCATION' 'INTEGER' X,Y);

This procedure exchanges the variables whose locations are passed as the parameters X and Y.

'PROCEDURE' SWOPOPT;

This procedure is called before the application of reversible operators, to reverse the order of the two Arithmetic Operands on top of the stack, if this would reduce the amount of code subsequently generated. The aim is to have the LH operand in the preferred (or any) accumulator, and to avoid having to unpack or rescale the RH operand.


Storeaway
is assignment dummy?
Yes
Delete R.H.S. from stack
Exit
No
Calculate field optimisation flag (FFLAG)
Calculate address to be stored in accumulator record (ADD)
Select accumulator (ACC)
Output instruction to copy R. H. S. into accumulator if required
Output trace instructions if required
Is assignment to partword?
Yes
Is assignment of constant?
Yes
No
No
Output store or add/subtract from store instruction
Output instructions to:
shift
mask
exchange
mask
add back
Shift and mask constant. Output instructions to:
copy LHS
mask
insert constant
store
Delete record of copies of LHS
Delete RNS from stack
Update accumulator record
Exit


SUBTERM
IS TERM A CONSTANT?
Yes
IS OPERATION ADD?
No
Yes
ADD CONSTANT TO DIRECT ADDRESS OF VARIABLE
SUBTRACT CONSTANT FROM DIRECT ADDRESS OF VARIABLE
No
IS INDIRECT ADDRESS OF VARIABLE NULL?
No
Yes
DOES TERM REQUIRE EVALUATION OR IS OPERATION SUBTRACT?
Yes
EVALUATE TERM IN MODIFIER AND INSERT MODIFIER NUMBER IN INDIRECT ADDRESS OF VARIABLE
No
IS TERM FLAGGED AS LOCATION
Yes
ADD ADDRESS OF TERM TO DIRECT ADDRESS OF VARIABLE
No
INSERT ADDRESS OF TERM INTO INDIRECT ADDRESS OF VARIABLE
DOES TERM REQUIRE UNPACKING
Yes
UNPACK TERM
No
IS TERM IN MODIFIER AND OPERATION ADD?
Yes
EXCHANGE DIRECT ADDRESS OF TERM AND INDIRECT ADDRESS OF VARIABLE
No
SELECT MODIFIER FOR INDIRECT ADDRESS OF VARIABLE
IS OPERATION ADD?
Yes
No
OUTPUT INSTRUCTION TO SUBTRACT TERM FROM MODIFIER
OUTPUT INSTRUCTION TO ADD TERM TO MODIFIER
INSERT MODIFIER NUMBER IN INDIRECT ADDRESS OF VARIABLE
REMOVE TERM FROM STACK AND UPDATE ACCUMULATOR RECORD
RETURN

```
'INTEGER' 'PROCEDURE' TESTSTRING
                    ('VALUE' 'INTEGER' S1,S2);
```

This procedure compares two strings, pointers to which are passed as the parameters S1 and S2. If the strings are identical the result of the procedure is zero, if they are not identical the result is non zero.

```
'PROCEDURE' TEXT
          ('VALUE' 'INTEGER' STRING);
```

This procedure outputs the string whose address is given by the parameter STRING. The length of the string is obtained by use of the procedure NEXTCHAR, the first character of the string being its length. The characters forming the string are obtained by the repeated use of NEXTCHAR and output by use of the procedure OUTCHAR. It should be noted that the use of NEXTCHAR alters the value of the parameter, STRING. This does not of course alter the actual parameter passed from the point of call, or the string itself, but merely the local value of the parameter. Zero length strings cause no characters to be output.

```
'PROCEDURE' TEXTLINE
          ('VALUE' 'INTEGER' S);
```

This procedure outputs the string S, followed by a newline.

```
'PROCEDURE' THEAD
```

This procedure outputs ten inches of blank tape on the punch, using PUNCHLOOP. It also resets the checksum for the characters output on the tape, TSUM, to zero.

```
'PROCEDURE' TOPACC
          ('VALUE' 'INTEGER' ACC);
```

This procedure is used to update the Arithmetic Operand on the top of the stack (i.e. the RH operand), to show that it is now held in the accumulator whose number is given in ACC. This is performed by means of a call of INACC.

```
'INTEGER' 'PROCEDURE' TOSADD;
```

The result of this procedure is the direct address of the top Arithmetic Operand on the stack, which is deleted.

```
'PROCEDURE' TRACESTRING
          ('VALUE' 'INTEGER' LNO,STRING);
```

This procedure is used to set up calls on trace procedures having a string as a parameter. The library reference number is passed as the parameter LNO, and the compile time address of the string as the parameter STRING. The call of the trace procedure is output using LIBTRACE and then the string is output, destined for the special constants area (Special), using DOSTRING with OUT24 as a parameter.

'PROCEDURE' TTAIL;

This procedure outputs a tail on paper tape. This consists of ten inches of blank tape, a stop directive (7STOP), twenty erases, and another ten inches of blank tape.

'PROCEDURE' UJBACK;

This procedure outputs an unconditional jump using the variable BJA (back jump address) as the address. This is output where it is required to jump back and repeat after testing the condition for continuation.

'PROCEDURE' UNARYMINUS;

This procedure is called where it is required to negate the Arithmetic operand on the top of the stack. If it is a constant then it is negated within the compiler. Otherwise instructions are output to negate it and leave the result in an accumulator. This procedure is called from the syntax, and also from SELOPTA,SUBA, and SIMPLEASS in the case of a unary negation operator; and also from CONDMINUS and ZEROCOMP when compiling conditions.

'INTEGER' 'PROCEDURE' USELAB

('VALUE' 'INTEGER' LAB);

This procedure is called, where it is required to use the address of the label whose record is specified by LAB, as the address part of an instruction to be output. The result is either the address of the label, if it is set, or the chain address of the label if it is not. If this is the first reference to the label in the current block then the result is zero, but this requires no special action at the point of call. If the label is unset then the label record is updated, to include the instruction about to be output, on the chain for the label.

'PROCEDURE' WARN

('VALUE' 'INTEGER' TA,TB);

This procedure is called in cases where a recoverable error, but of insufficient gravity to prevent the compiled program from running safely, is discovered. This procedure output the two strings TA and TB, if non zero, separated by " : ". This is followed by a newline, and PRINTBUFF is called to print out the contents of the input buffer.

'PROCEDURE' XSPARAMS

('VALUE' 'INTEGER' PTR);

This procedure is called if the procedure identifier whose declaration record is pointed at by PTR is specified to have more than the number of parameters allowed by the implementation. After the sixth parameter, IDTYPE is marked negative. If there is a seventh parameter, a call is made to XSPARAMS which prints out a warning message; and also clears IDTYPE so that no further calls are made for the eighth and subsequent parameters.

'PROCEDURE' ZEROACCS;

This procedure is called to clear the array ACCS, which holds a memory of the current contents of the accumulators of the object program. This is necessary where there is a flow of control which invalidates the contents of ACCS.

Label ABSADD:

This section is called in 'ABSOLUTE' communicators, after the integer constant specifying the absolute address for the preceeding identifier has been processed. This address is inserted into the direct address field of the identifier record being formed at the base of the stack. The address is masked to ensure that it does not overflow into the tag field, which in this case is zero.

Label ADD:

This section is called to output instructions for the addition of two terms. This is performed by means of a call of ADDSUB with the function code 02 as the parameter. As this function may be applied with its operands in either order, a call is first made to SWOPOPT for optimisation purposes.

Label ADDA:

This section is called after the first term of the expression on the right hand side of an assignment symbol, if the term is unsigned or preceded by a + symbol. If the term is a variable, and is the same variable as the one on the left of the assignment symbol, and is not a partword or of type floating, ASSFUN is set to function 12, so that an "add-to-store" assignment will be used. This section is not called if assignment trace is required.

Label ADDRSPEC:

This section is called to insert the current special data address into the direct address of the identifier record being prepared at the base of the stack. This is used in all cases of 'SPECIAL' 'ARRAY's, and also for 'COMMON' 'LABEL's, 'SWITCH'es, and 'PROCEDURE's.

Label ADDRSW:

This section is called after the occurrence of the symbol 'SWITCH' in a switch declaration. The address of the (virtual) zeroth element of the switch is calculated, and stored in the base of the stack in preparation for generating the identifier record. The variable TOPTEN is cleared to ensure that the top ten bits of each entry in the switch list are clear.

Label ANSCHECK:

This section is called at the end of an 'ANSWER' statement, to ensure that the expression is evaluated to the required type and scale, given by EXPSCALE, in accumulator 7. Before returning to the syntax analyser via BEHEADEXIT, to delete the Arithmetic Operand record of the expression; STATUS is set to 2 to indicate that the exit instruction has not yet been output.

Label ANYPREF:

This section is called to select an accumulator for the evaluation of Comparisons within Conditions. An accumulator number is supplied by FINDACC, and stored in PREFACC.

Label ASSCV:

This section is called after the (first) expression of a for element. CCC is set to the TYPEBITS of this expression, only the sign bit (constant marker) being relevant. The instructions to assign this expression to the control variable are output by means of a call of STOREAWAY with a parameter of zero, as trace is not required at this point. If this is the second for element, and the first was simply an expression, FORSTATE will have a value of 2. In this case, if the next symbol is not 'WHILE', a call is made of SETRT, to cause the controlled statement to be treated as an anonymous procedure; and FORSTATE is reset to 4. (If the following symbol is 'WHILE', FORSTATE will be set to 3 by WHILEL.) In order to avoid the inefficiency of treating the controlled statement as a procedure in cases where the for list is of the form:

Expression Comma Expression While

if the first element is an expression, the decision to treat the controlled statement as an anonymous procedure is delayed until after the expression of the second element.

Label ASSTRACE:

This section sets 'BIT'[3]TRACE to the value of ACCUMULATOR. It is called in the case of a 'NO' 'ASSIGNMENT' 'TRACE' directive, in which case the value of ACCUMULATOR will be zero; and also in the case of an 'ASSIGNMENT' 'TRACE' directive, in which case the value of ACCUMULATOR will be 1.

Label AYMCHECK:

This section is used to check that an identifier, which is used without a subscript, does not require one. If it does, a warning message is output.

Label BEGINPROC:

This section is called at the start of a procedure declaration. If the procedure body requires a "jump round" this is output, or if there is a defered 'ANSWER' action, this is completed. PROCSTACK is called to perform housekeeping on the variables used for procedure declarations. The LINK address is set to the data location which had been allocated to the procedure identifier record, this direct address entry is changed to the current value of the Special Data allocation address, DUECHAIN being also set to this address. This location, which will be used to hold the entry point of the procedure, is allocated, and cleared by OUT24CS. If procedure trace is required, the procedure identifier string is output, destined for the Special Data area, by SPECSTRING, and its address stored in PROCSTRING, which acts as a trace flag.

Label BEGINPROG:

This section is called at the start of a program segment. The heading for the paper tape is output by STARTSEG, and STATUS and DUECHAIN are set to zero. Thus it is expected that a program segment will be entered at its first instruction, even though this will normally be an unconditional jump around procedure bodies. BEGINBLOCK is called to set up an outer block level for the program.

Label BEGINPSPEC:

This section is used to initiate the creation of a parameter specification record of a procedure specified in a communicator, or a procedure parameter declared within a program or Library segment. Calls are made on MAKEPSPEC and EXCPSPEC to accomplish this. If the procedure is a 'VALUE' procedure, a call is made on NEXTPARAM, and a jump made to NEXTPSET, to set up the first (context parameter) entry inthe record.

Label BEHEADEXIT:

This section is used by a number of compiling actions to remove the top Arithmetic Operand record from the compiler's main stack, before returning to the syntax analyser. This record, which is no longer required, is removed by calling BEHEAD. When the record is removed, the operand pointers LH and RH are updated by SETLHRH.

Label BITDECFAIL:

This section is used by NOBITS,NOAFTER,FIELDPOSN, and UNSFIELD if the values of the integers used in field specifications exceed the allowable range. This section prints a warning message, and returns control to the syntax analyser.

Label BITSIN:

This section is called where a 'BITS' operation is used in an expression, before the expression to which the operation is to be applied has been processed. This section packs the partword and type information, held in BITSPEC and IDTYPE, into BITSHIFT; as these variables may be used during the following Expression. The information packed in BITSHIFT will subsequently be used by RHSBITS.

Label CALLPROC:

This section is called when the appropriate number of parameters of a procedure call have been processed. This number may be zero. Starting from the position on the stack given by FPP, there will be a five word Arithmetic Operand record for each parameter. The types of these will have been checked by ANYMORE, and they will all have zero indirect addresses. Thus the actual object to be passed will either be in an accumulator, or obtainable by means of a call of PICK, which may involve unpacking, but which will not require the use of a modifier.

Label CALLPROC:/

Each record is examined in turn, (left to right) and the number of the accumulator required, obtained from the PAC field. If this accumulator is already holding another parameter, an instruction is output to exchange this with the accumulator in which the other parameter is required to be passed. This is repeated until the accumulator specified in the current record is either free, or contains the required parameter. A call of PICK is made to copy the parameter from store, if required, and to unpack if necessary.

After this scan has been completed, the required parameters have been placed in the required accumulators with the minimum number of instructions. The call of the procedure is output by OUT27, and ZEROACCS called to clear the accumulator record ACCS; as it is assumed that the contents of all the accumulators (except 7 if typed procedure) will be undefined on return from the procedure.

Label CHECKCV:

This section is called after the control variable of a for statement has been processed. If the variable is a partword, or is not of type integer or fixed, or is an unsubscripted array identifier, an error message is output, and its type changed to integer to allow compilation to continue. The exclusion of floating point control variables is an implementation restriction. In this case it is always faster to increment and test a fixed point variable, and subsequently float it each time, than it is to carry out the whole operation in floating point arithmetic. It also requires fewer instructions, and is often more accurate, as it avoids cumulative round-off and truncation errors.

After checking the type of the control variable, this type and scale is stored in EXPSCALE, and SCALEFIRM set, to ensure that the expressions of the for list are evaluated to the type and scale of the control variable. PREFACC is set to an arbitrary number, and ASSFUN set to function 10. This may subsequently be modified by assignment optimisation. A pointer to the Arithmetic Operand record of the control variable is placed in CV, and a call made of DIAG to output diagnostic information. This section returns to the syntax analyser via FORESTORE, to store the index expression of the control variable, if one has been evaluated.

Label CLEARTYPE:

This section is called to clear the variable IDTYPE, prior to assembling type and scale information, in declarations, specifications, and typed primaries.

Label CODELAB:

This section is called after the occurrence of a label preceding an instruction in code. This action is performed by calling FIXLABEL with the result of a call of LOOKUPLAB as the parameter. This is similar in action to SETLAB, but does not generate any label trace instructions, even if 'LABEL' 'TRACE' is specified at this block level.

Label CODESHIFT:

This section is called to set up an anonymous Arithmetic Operand for a shift instruction in code. A call is made to NONAME to set this up, and the commentary of this record changed to "(SHIFT)". This section returns to the syntax analyser via INCTOS which deals with any integer constant specifying the number of places of shift.

Label COMOFF:

This section is called at the end of a 'COMMON' segment. A check is first made to ensure that the (hopefully) unique checksum for this common segment is not zero, this checksum then being output to the loader. The output is completed by a call of FINISHSEG, and the COMMON flag cleared. A scan is then made of the DECLIST to re-address the common entries, and to output a common-check tape. Whilst a 'COMMON' segment is being compiled, all addresses within it are allocated with respect to its Data and Special Data areas (tags D and S). When the size of these are known, the addresses may then be relocated relative to the start of the segment (tag C). N.B. The compiler assumes that the Special Data area of a common segment immediately follows the Data area.

Label COMON:

This section is called at the start of a 'COMMON' communicator segment. A check is first made to ensure that no other common specifications exist for this run of the compiler; program segments being only allowed to be compiled using one common segment. After this check, the COMMON flag is set, and the heading output by STARTSEG.

Label CONDADD:

This section is called after the second, and each subsequent, term of the expression on the right hand side of a comparator has been processed; if the term is preceded by a + symbol.

In the interests of efficiency, the expressions on either side of the comparator are evaluated as one expression and the result tested against zero. Provided that the comparator is neither > nor <=, the Terms on the right which are preceded by a + sign are subtracted, and those which are preceded by a - sign are added. Otherwise the left hand side is negated, and the expected add and subtract operations used.

This section jumps to either ADD or SUB depending on the comparator used.

Label CONDMINUS:

This section is called after the first Term of the expression on the right hand side of a comparator has been processed; if the Term is preceded by a - sign. In the interests of efficiency, the expressions on either side of a comparator are evaluated as one expression, and the result tested against zero.

If the comparator is neither > nor <=, this section jumps to ADD to output instructions to add this term to the expression.

(Addition being taken as being equivalent to negation and subtraction)

Otherwise the expression on the left is negated by means of a call of UNARYMINUS, the operand records interchanged by PERM, and a jump made to SUB.

Label CONDPLUS:

This section is called after the first Term of the expression on the right hand side of a comparator has been processed; if the Term is not preceded by a - sign. In the interests of efficiency, the expressions on either side of a comparator are evaluated as one expression, and the result tested against zero.

If the comparator is > or <= PERM is called, to exchange the Arithmetic Operand records of the left hand expression, and the right hand term. This changes the type of test required to one which is available in the machines instruction set. A jump to SUB is made, to output instructions to subtract the two operands.

Label CONDSUB:

This section is called after the second, and each subsequent, Term of the expression on the right hand side of a comparator has been processed; if the Term is preceded by a - sign.

(See CONDADD above)

Label CONSTANT:

This section is called where a numeric constant occurs in an expression, and also as an operand in code. It sets up an Arithmetic Operand record on the stack. The commentary is set to "(CONST)", and the TYPEBITS word set to the type of the constant, NUMBERSCALE, which has the sign bit already set. This denotes that the operand is a constant whose value is stored in the DIRADD field of the record.

Label DECSIZE:

This section is called after declarations of single word variables, to record the number of Data locations required.

Label DIVIDE:

This section is called to output the instructions for the division operation. The scale and type to which a division operation is carried out depends on the scales and types of the operands, and also the context in which the division occurs. The local procedure ENVTEST is used for testing the "environment" of the division. If the division occurs as the last operator in a Term occurring in an expression which is required to be evaluated to a specified scale, or the proposed scale SCALE has fewer integer bits than the current value of the expression scale EXPSCALE this returns a result of 1, otherwise 0.

Before the calculation of the scale to which the division will be performed is carried out, a check is made on the operands. A call is made to ISINACC in case there is a copy of the left hand operand in an accumulator. If the right hand operand is a location, or requires unpacking, a call is made on PICK to copy it into an accumulator. The syntax of this implementation specifically prohibits multiplication or division of addresses, but this may still be performed, if the address is within a bracketed expression.

Label DIVIDE/

If either of the operands, or EXPSCALE, is of type floating, then the division is carried out in floating point arithmetic, by means of a call of FLOATOP, with a parameter of 4.

If the right hand operand is an integer constant which is a power of two, the division will not be performed using a divide instruction. If the left hand operand is of type integer, this will be performed by shifting, which will be combined with any shifts required for unpacking and rescaling. If ENVTEST returns a zero answer, the result is evaluated to integer scale, otherwise the integer is rescaled to the (fixed) scale given in EXPSCALE. If the left hand operand is of type fixed, the division is carried out by adjusting the scale field of its operand record. In this case the number of significant bits remains unchanged.

If both operands are of type integer, the division will be carried out as an integer division, unless ENVTEST indicates that there is an overriding scale requirement. The number of significant bits of the result will not be specified unless both the operands have the number of significant bits specified. In this case the number of significant bits is given by the number of significant bits of the left hand operand minus the number of significant bits of the right hand operand plus one to make allowance for the range of the result, plus one for the sign bit. If this is negative, it is set to zero (unspecified) and is otherwise subject to a limit of 24.

If the left hand operand is of type fixed, and the right hand operand of type integer, the division is evaluated to the scale of the left hand side, unless ENVTEST indicates that there is an overriding scale requirement or the integer is a constant. In this case the integer constant is converted to the equivalent fixed constant, and the division performed as a division of a fixed by a fixed. The use of the left hand scale is based on the assumption that the integer will be small. If this is not the case, the division should be bracketed and typed.

If the left hand operand is of type integer, and the right hand operand of type fixed, and there is no overriding scale requirement, the scale to which the division will be performed is based on the scale obtained if the left hand operand were to be converted to fixed, and the division treated as a division of a fixed by a fixed. The rescaling is not performed at this point.

The final case is the division of a fixed by a fixed. Unless ENVTEST indicates that EXPSCALE should be used, the scale of the result has the smaller number of significant bits of the two operands, and the number of integer bits is the number of integer bits of the left hand operand minus the number of integer bits of the right hand operand. In a simple case of division this will not give rise to rescaling.

When the scale to which the division will be performed has been decided, the number of shifts right prior to the division, SHIFTS, is calculated. If this is less than 23, an instruction is output to clear the auxiliary register. If the number is negative (shift left required), this is combined with any rescaling and unpacking required, by means of a call of PICK. Otherwise a call is made of GOODPICK to ensure that the left hand operand is in an accumulator, this being followed by a separate right shift instruction. In certain cases this may lead to slightly inefficient code.

Label DIVIDE/

The instruction to perform the actual division is then output. If the right hand operand is in the preferred accumulator the two operand records are now interchanged by PERM. The final instruction output is to copy the result of the division, which is in the auxiliary register, into the accumulator whose number is given in the left hand Arithmetic Operand record. This section returns to the syntax analyser via BEHEADEXIT, which removes the top Arithmetic Operand record from the stack.

Label DOCS:

This section is called after the occurrence of the symbol 'DO', which terminates the Forlist of a Forstatement. The value of FORSTATE, on entry, indicates the type of the last element and the complexity, of the Forlist. If FORSTATE is greater than three, the controlled statement is to be treated as an anonymous procedure, and instructions will be required to store, and exit on, the link.

This section outputs any instructions required for the testing of step-until elements, and for the calling of, or jumping round, the controlled statement. It also reserves any locations required for

1 The index of the control variable, if subscripted.

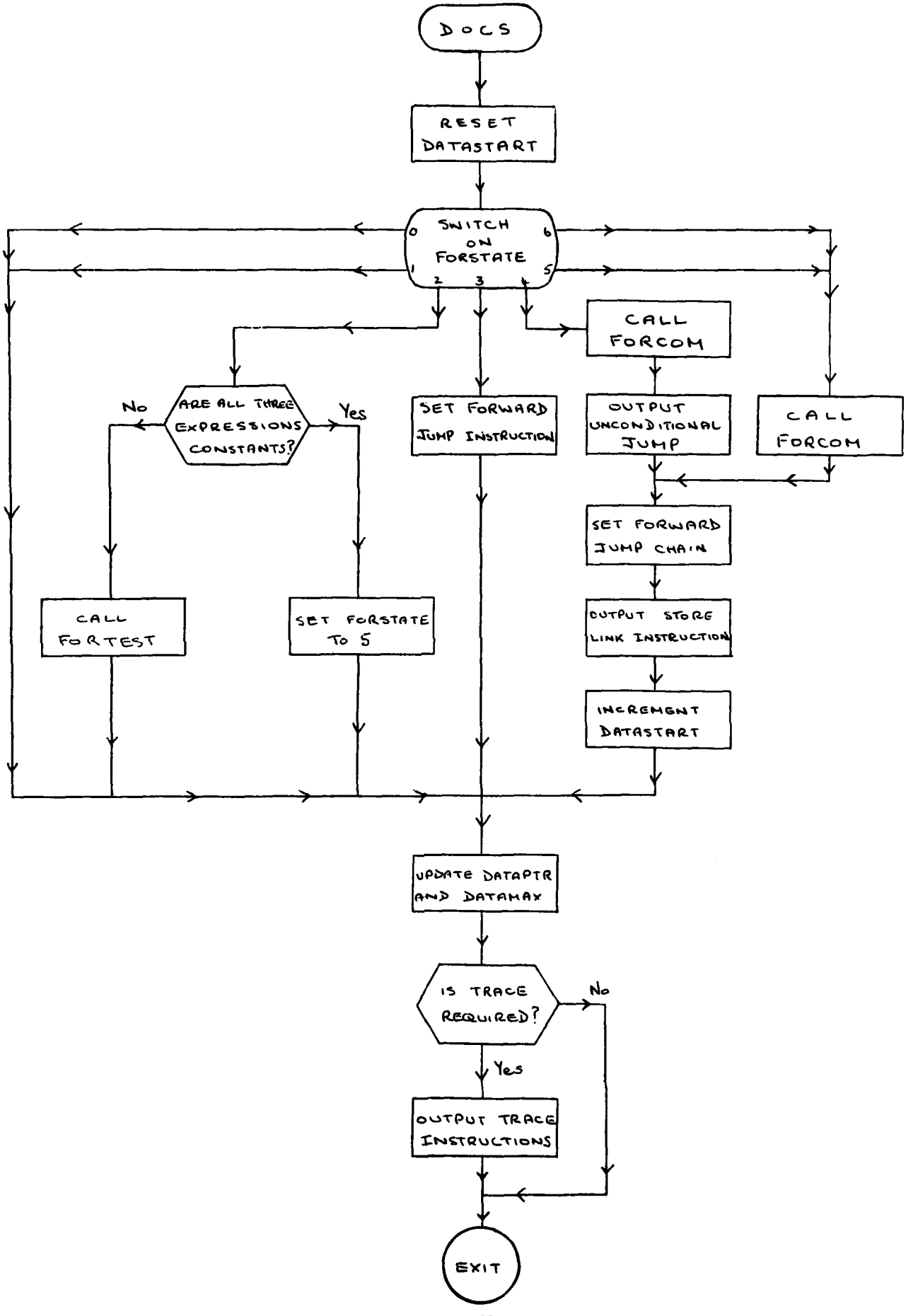
2 Holding step and limit values, if they have been evaluated.

3 The link, if the controlled statement is treated as an anonymous procedure.

Any other workspace, temporarily used in the evaluation of expressions in the Forlist, is relinquished. DATASTART,DATAPTR, and DATAMAX are updated as required. The locations reserved will be returned at the end of the controlled statement, and the above variables reset. If 'LOOP' 'TRACE' is required, a call of L9 is generated, with the value, type, and identifier string, of the control variable as parameters.

If the value of FORSTATE is greater than four, on entry, it will be set to four by FORCOM. The value of five is then used to indicate that the special case of a single step-until element, with three constants, has been detected. This will result in the test and incrementation of the control variable taking place at the end of the controlled statement, and consequently more efficient code being generated. A flowchart for this section is given, and the values of FORSTATE and the corresponding of Forlist is tabulated below:

FORSTATE Entry	FORSTATE Exit	
0	0	Expression
1	1	Expression 'WHILE'
2	2 or 5	.. 'STEP' .. 'UNTIL' ..
3	3	Expression,Expression 'WHILE'
4	4	.... , Expression
5	4	.... , Expression 'WHILE' (Excluding case 3)
6	4	.... , .. 'STEP' .. 'UNTIL' ..


DOCS
RESET DATASTART
SWITCH ON FORSTATE
0
1
2
3
4
5
6
CALL FORCOM
ARE ALL THREE EXPRESSIONS CONSTANTS?
No
Yes
SET FORWARD JUMP INSTRUCTION
OUTPUT UNCONDITIONAL JUMP
CALL FORCOM
SET FORWARD JUMP CHAIN
CALL FORTEST
SET FORSTATE TO 5
OUTPUT STORE LINK INSTRUCTION
INCREMENT DATASTART
UPDATE DATAPTR AND DATAMAX
IS TRACE REQUIRED?
No
Yes
OUTPUT TRACE INSTRUCTIONS
EXIT

Label DOSHIFT:

This section is called to apply shift operators ('SRA','SLA','SRL', and 'SLC') using the top two Arithmetic Operands on the stack. Before this can be accomplished, a call has to be made to KILLEXPR to remove the expression level set up for the optimisation of the expression giving the number of places of shift. The instructions for the shift operation are then output by OPERATE, which is passed the required function, held in BITSHIFT, as its parameter.

Label ELSEFI:

This section is called after both the consequence and the alternative expressions of a conditional expression. It ensures that each is evaluated to the scale given by EXPSCALE, in the accumulator given by PREFACC. A call is made to REVCHAIN so that the chain of the jump to skip over the expression will be set by the next action. (ELSEX or FIEX)

Label ELSES:

This section is called following the 'ELSE' in a conditional statement. A call is made on REVCHAIN, as the chain which holds the jumps to skip over the first part of the 'IF' statement is now due to be set. If STATUS is 4, then the statement between the 'THEN' and the 'ELSE' is void, and the condition is reversed using REVCJ.

If STATUS is 3, the statement is of the form:

'IF' . . 'THEN' 'GOTO' label 'ELSE'

in this case a jump is not required to skip over the following statement; STATUS being set to 0 to indicate this.

If STATUS is 0, the statement preceding the 'ELSE' is a normal statement, and a skip is required over the following statement; STATUS being set to 3 to indicate this.

The remaining two cases, STATUS of 1 and 2, require no action at this point.

Label ELSEX:

This section is called following the occurrence of the symbol 'ELSE' in a conditional expression. It outputs a jump around the alternative expression, by means of a call of OUTUJ. The chain from the jump(s) around the consequence expression is set by SETDUE.

SCALEFIRM is set to ensure that the alternative expression is evaluated to the same scale as the consequence expression. This section returns to the syntax analyser via BEHEADEXIT, to remove the Arithmetic Operand record for the consequence expression from the stack.

Label ENDARRAY:

This section is called after the end of a set of arrays having common bounds. This may be followed by a further set. If the declaration is not overlayed, the total data requirement for the arrays, which is the product of the number of arrays and the sizes of all the dimensions, is given in ARRAYS, which is used to increment DATAMAX. If the declaration is overlayed, OVERBASE is incremented by ARRAYS.

Label ENDDECS:

This Section is called after the end of the declarations at the head of a block. DATAPTR and DATASTART are set to the current data allocation address held in DATAMAX. These variables will be used for the allocation of anonymous workspace for temporarily dumping the contents of accumulators. DATAPTR will contain the next address to be allocated, and will be reset between statements; its maximum value being recorded in DATAMAX. After the declarations at the head of a block, normal presetting (if allowed at this level) is no longer allowed, and the flag PRESETOK is cleared.

Label ENDFOR:

This section is called after the completion of the controlled statement, to output the instructions required to complete the loop. The action required depends on the value of FORSTATE set up by DOCS, which indicates the type of the last element and the complexity of the Forlist.

If FORSTATE is one or three ('WHILE'), an unconditional jump back, to repeat the assignment and test the condition, is output. If however STATUS is non-zero, this jump is not required, and is suppressed. Any chain due to be set after the controlled statement is set to the repetition address, held in BJA.

If FORSTATE is two ('STEP' 'UNTIL'), the instructions are output to increment the control variable and to jump back to the test against the limit.

If FORSTATE is four the controlled statement is treated as an anonymous procedure, and an instruction to return using the stored link is output.

If FORSTATE is five, this indicates the special case of 'STEP' 'UNTIL' with three constants. If a copy of the control variable does not exist in an accumulator, then an instruction is output to copy it into accumulator 7. Instructions are then output to add the step value to the control variable and to store the result. An instruction is then output to subtract the limit value, if non zero, and a test instruction output. The sense of this test instruction depends on the sign of the step. If the step is positive, a jump if zero or negative is required. As this is not available, the limit has one added to it, so that the zero case is included in the negative test. The final part, common to all cases, completes the housekeeping for this level of 'FOR' statement. The SKIPCHAIN, which is used by the final test, is copied into DUECHAIN, and SKIPCHAIN reset to its value before the statement, which is held in COPYSKIP. DATASTART is reset to release any anonymous locations used, and the previous values of any enclosing 'FOR' statement reset by means of a call of OFFSTACK. Finally, a call is made to ENDLABBLOCK, to remove the label block level which prevented illegal entry to the controlled statement.

Label ENDPROC:

This section is called at the end of the body of a procedure declaration, in program and library segments.

EXCPSPEC is called to prepare for the completion of its parameter specification record, and STATUS is set to 1 to indicate that the following instruction is not directly entered. LOCALLIMIT is reset to point at the procedure identifier record, this value being set into DECLIST by the call of ENDBLOCK, to complete the block forming the procedure body. The values of the eight locations used by procedure declarations, and chained up by PROCCHAIN, are restored by means of a call of OFFSTACK, and finally the parameter specification record is completed by means of a call of FINISHPSPEC. It should be noted that once a procedure body has been compiled, only its parameter specification record remains, and not the identifier records of its parameters.

Label ENDPROG:

This section is called before the final 'END' symbol of a program segment. If the program does not end with a jump, but "drops off the bottom", a call of L1 is generated.

A scan is then made of all the communicator specification records. This detects entries required to be inserted in the 'COMMON' segment, and labels used which are external to this program segment. In this latter case, the label is set to the current program address, and an indirect jump to the label output. ENDBLOCK is then called to perform the normal end of block actions. After this call, DATAMAX will hold the total data requirement for the segment, and LABDECLIST will point at a list of any unset labels. These are dealt with by the call of FINISHSEG, which also completes the paper tape output for this segment. A flowchart is given for this section.

Label ENDSPEC:

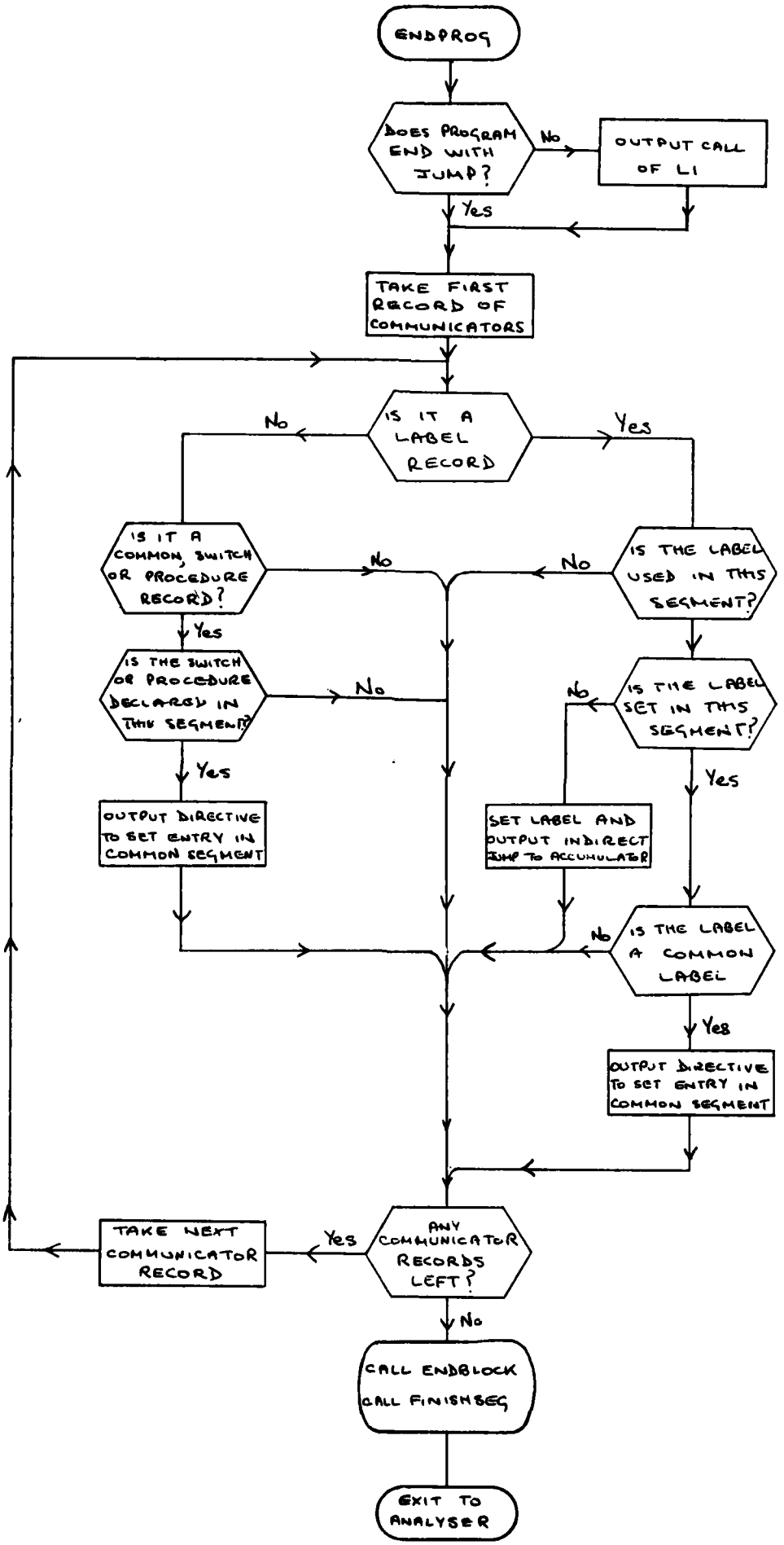
This section is used to terminate the generation of a parameter specification record of a procedure specification. This is accomplished by means of calls of EXCPSPEC and FINISHPSPEC. This resets the values of IDTYPE and PARAMPTR, which will be required if the procedure (parameter) specification forms part of a procedure declaration.

Label ENDST:

This section is called at the end of each statement compiled, to reset the variable DATAPTR, used for the allocation of anonymous workspace, to its "home" value, held in DATASTART. As anonymous workspace is not required to be carried from one statement to the next, it may be re-used in following statement.

Label EXITCHECK:

This section is called at the end of compilation of a non-code procedure. If the procedure is untyped, i.e. does not deliver a result directly, a check is made to test whether the last instruction of the procedure was an unconditional jump, and that a return jump is not required. If this is the case no action is taken by this section; otherwise a call is made on SETDUE to set any chains. If, on the other hand, the procedure is typed, and is therefore required to deliver an answer; a check is made


ENDPROG
DOES PROGRAM END WITH JUMP?
No
OUTPUT CALL OF L1
Yes
TAKE FIRST RECORD OF COMMUNICATORS
IS IT A LABEL RECORD
No
Yes
IS IT A COMMON, SWITCH OR PROCEDURE RECORD?
No
No
IS THE LABEL USED IN THIS SEGMENT?
Yes
IS THE SWITCH OR PROCEDURE DECLARED IN THIS SEGMENT?
No
No
IS THE LABEL SET IN THIS SEGMENT?
Yes
Yes
OUTPUT DIRECTIVE TO SET ENTRY IN COMMON SEGMENT
SET LABEL AND OUTPUT INDIRECT JUMP TO ACCUMULATOR
No
IS THE LABEL A COMMON LABEL
Yes
OUTPUT DIRECTIVE TO SET ENTRY IN COMMON SEGMENT
TAKE NEXT COMMUNICATOR RECORD
Yes
ANY COMMUNICATOR RECORDS LEFT?
No
CALL ENDBLOCK
CALL FINISHSEG
EXIT TO ANALYSER

Label EXITCHECK/

to ensure that the procedure contained at least one 'ANSWER' statement, and that the last effective statement of the procedure was either a 'GOTO' or an 'ANSWER'. If this is not the case, an error message is output. If the procedure is typed, and trace is required, a call is made on L6 with the result and its scale as parameters. In the case of a 'VALUE' procedure, the scale is obtained dynamically from the first (context) parameter, which must therefore not be altered by the procedure. If there is more than one 'ANSWER' statement, these will be chained up using EXITCH, which is set to the call of L6. Irrespective of whether the procedure is typed, if trace is required, a call on L5 is output, with the procedure identifier string as the parameter. It is assumed, in the case of typed procedures, that L5 will reset @7 to the value supplied to L6. Finally, an exit instruction is output, using LINK as the address.

Label EXPRTYPE:

This section is called after the completion of a typed bracketed expression, to ensure that the expression is evaluated to the specified type and scale. This may not have already been performed if the expression did not contain addition or subtraction operations. The expression is evaluated to the required scale by means of a call of PICK. The section OFFSTEXPR is then entered to remove the expression level from the stack.

Label EXTADD:

This section is called, in 'EXTERNAL' communicators, after the integer constant(s) specifying the external relocator (and displacement) have been processed. The fourteen bit address is formed from the five bit relocator number, and the nine bit displacement to which is added 256. This address is masked, and the external tag(6) added. The address is then inserted into the direct address field of the identifier record being formed at the base of the stack.

Label FI:

This section is called at the end of a conditional statement, irrespective of whether it had an 'ELSE' part. If STATUS is 4, then the statement is void, and the test instruction is only output if FUNCTION is 24, the statement being of the form:

'IF' 'OVERFLOW' 'THEN' ;

which may be used to clear overflow. If STATUS is 3 or 4, the 'ELSE' part has been omitted, and STATUS is set to 0.

The SKIPCHAIN is joined to the DUECHAIN, if required, by means of a call of JOINCHAINS. The stacked value of SKIPCHAIN, chained up with IFCHAIN, is restored to its value prior to the 'IF'.

Label FIELDDISP:

This section is called after the integer giving the Wordposition of a table field has been processed. The address of the table field is calculated and stored at the base of the stack in preparation for the identifier record for the field being moved onto the main stack declaration list by NEWNAME.

Label FIELDPOSN:

This section is called when the integer specifying the starting bit position (Bitposition) of a partword is processed. This integer, together with the result held in BITSPEC allows the two shift fields of the partword descriptor to be calculated. These fields are packed in BITSPEC, and are respectively:

MSS (most significant shift)
shifts to align field to top of word

LSS (least significant shift)
shifts to align field to bottom of word
and are calculated as follows:

MSS := Bitposition (of most significant bit)

LSS := 24 - Totalbits - Bitposition
using Ferranti bit numbering, sign bit = bit 0.

If, after the above calculations are performed, LSS is negative, the partword specification is illegal, i.e. a field has been specified which "hangs off" the word. In this case the error is reported by BITDECFAIL.

Label FIEX:

This section is called at the end of a conditional expression. The instructions to evaluate both expressions in PREFACC will have been output by ELSEFI. The chain holding the jump around the alternative expression is set by SETDUE. The variables of any enclosing condition are restored by OFFSTACK, and DUECHAIN is reset.

An Arithmetic Operand record for the result of the conditional expression is set up by a call of GRABVAR, and the TYPEBITS field set to EXPSCALE. The commentary is set to "(IFEX)", and a call made to TOPACC, with PREFACC as the parameter, to update the DIRADD field and also ACCS.

Label FINISHPROC:

This section is called after the call of a procedure has been generated, or after the final call of a multiple call. It reduces the procedure call level by a call of OFFSTACK. The PARAMSPEC field of the arithmetic operand record for the procedure is cleared, to avoid its interpretation as PARTWORD. This record and ACCS are updated by a call of TOPACC with a parameter of 7, to indicate that the result (if any) of the procedure is in accumulator 7. The TYPEBITS of this record will have been set by SETUPPROC. If the procedure is untyped, this record will be nonsensical, but will be immediately deleted by BEHEAD.

Label FIRSTDIM:

This section is called to process the first dimension of multi-dimensional arrays. ADDRARRAY is called to insert the addresses of the first level Iliffe vectors in the identifier records on the stack. The number of records to be processed is given in ARRAYS. After the call of ADDRARRAY the number of entries required in the first level Iliffe vector is given in ARRAYS, which has been multiplied by NUMBER in DODIM.

Label FORSTORE:

This section is entered from STEPUNT and from CHECKCV. It arranges to store in anonymous workspace the index of the control variable, if subscripted, and the values of the step and limit expressions if they are required to be evaluated. These are stored by means of a call of DUMPACCS, which is a brute force method of storing any information held in the accumulators. If an anonymous location is used by DUMPACCS, DATAPTR will be incremented. This value is copied into DATASTART so that this location will not be overwritten if space is required temporarily during the evaluation of the next expression. After each expression, DATAPTR is reset to the current value of DATASTART to free any location so used. RTL (routine link) is set to the maximum value attained by DATASTART, in case the controlled statement is to be used as an anonymous procedure. The current program address is recorded in BJA (back jump address), for use as a point to jump back to, in order to perform the required test before each repetition of the controlled statement. As a jump may be made to this point, a call is made of ZEROACCS.

Label FORTRACE:

This section sets 'BIT'[2]TRACE to the value of ACCUMULATOR. It is called in the case of a 'NO' 'LOOP' 'TRACE' directive, in which case the value of ACCUMULATOR will be zero: and also in the case of a 'LOOP' 'TRACE' directive, in which case the value of ACCUMULATOR will be 1.

Label GOTOL:

This section is called to process a 'GOTO' statement, where the destination is a label identifier and not a switch entry. The action depends on the value of STATUS, which is used as an index of the switch S. The actions taken in this section are designed to reduce as far as possible, the number of redudant unconditional jumps generated. These actions depend on the context in which the 'GOTO' statement occurs. (See 'PROCEDURE' STATUSCHECK). A flowchart is given for this section.

Label GOTOSK:

This section is called to output the jump instruction in the case of a 'GOTO' statement, where the Destination is a switch entry. This is performed by means of a call of INST with the RH arithmetic operand giving the address, and using function 27. Note that accumulator 4 is used, this is not necessary, but is in conformity with the other uses of function 27. The type checking of the switch identifier has previously been carried out by LABSK. The Arithmetic Operand Record created is treated in a similar manner to any other record, thus allowing the application of subscript optimisation. Before returning to the syntax analyser via BEHEADEXIT, to delete the record; STATUS is set to 1 to indicate that the following statement is not directly entered.

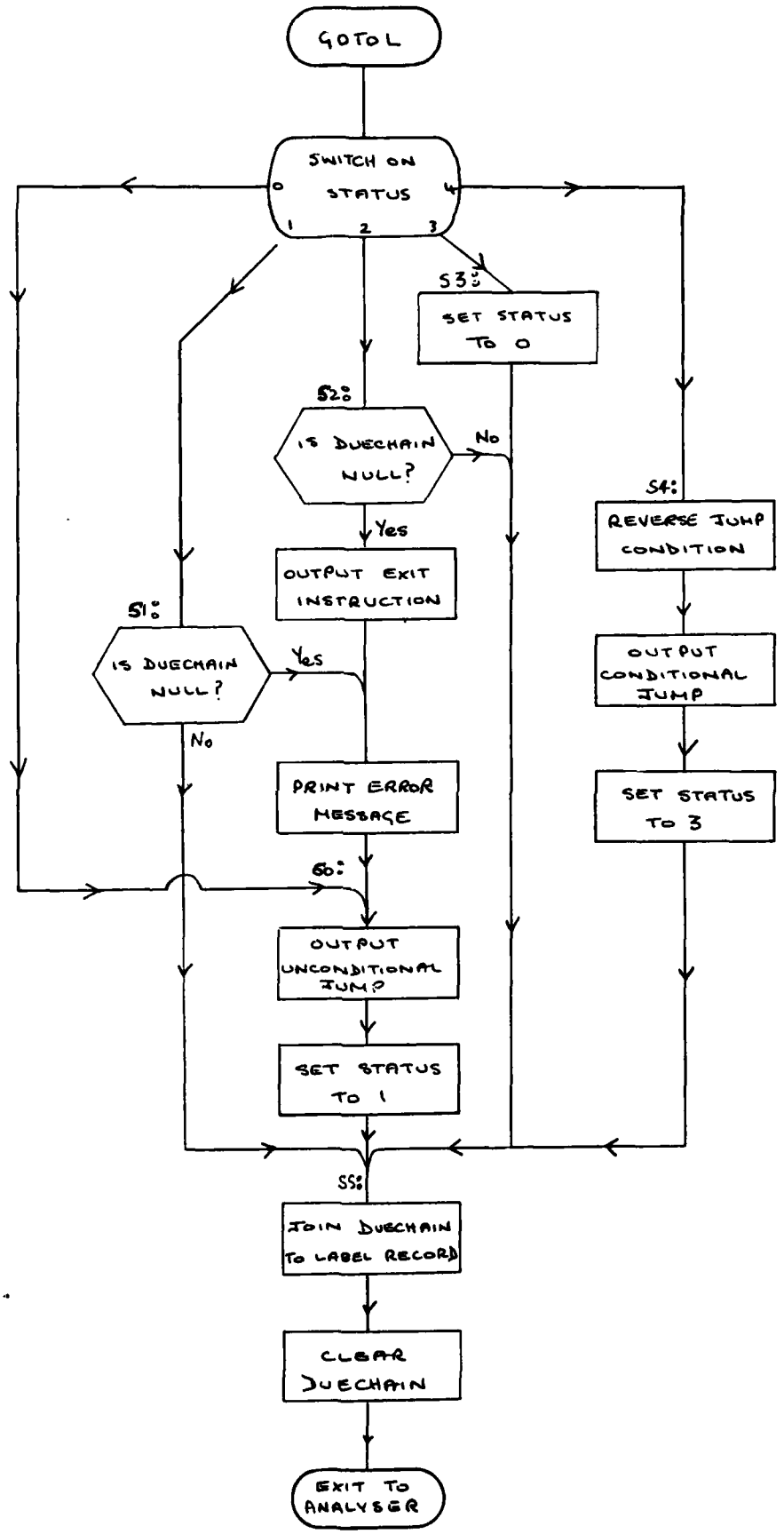
Label IFEX:

This section is called following the occurrence of the symbol 'IF' at the commencement of a conditional expression. Any partially evaluated results held only in accumulators are dumped by DUMPACCS. The chain DUECHAIN is stored in COPYDUE, and a call made of ONSTACK to stack:

IFCHAIN,SKIPCHAIN,COPYDUE and RELOP.

This is only required if the conditional expression occurs within another condition, but is carried out in all cases. DUECHAIN and SKIPCHAIN are then cleared in preparation for the following condition.

Where a conditional expression occurs in a context of defined scale, SCALEFIRM is non zero, and both the consequence and alternative expressions will be evaluated to this scale. Where the scale is not defined, the alternative expression will be evaluated to the final scale of the consequence expression. In either case, the expressions will be evaluated in the accumulator whose number is given in PREFACC.


GOTOL
SWITCH ON STATUS
0
1
2
3
4
S3:
SET STATUS TO 0
S2:
IS DUECHAIN NULL?
No
Yes
S4:
REVERSE JUMP CONDITION
OUTPUT EXIT INSTRUCTION
S1:
IS DUECHAIN NULL?
Yes
No
OUTPUT CONDITIONAL JUMP
PRINT ERROR MESSAGE
SET STATUS TO 3
S0:
OUTPUT UNCONDITIONAL JUMP
SET STATUS TO 1
S5:
JOIN DUECHAIN TO LABEL RECORD
CLEAR DUECHAIN
EXIT TO ANALYSER

Label IFS:

This section is called at the start of an 'IF' statement. A call is made on STATUSCHECK to complete any deferred actions, and to ensure that the statement can be entered. STATUS is then set to 4 to ensure completion of the following Condition.

The current SKIPCHAIN is stacked using IFCHAIN, and SKIPCHAIN set to zero. (DUECHAIN has been set to zero by STATUSCHECK). This clears the chains in readiness for the following Condition.

Label INCTOS:

This section is called to increment the direct address field of the Arithmetic Operand record of a code instruction by the integer constant held in NUMBER. This is performed by means of a call of ADDADD.

Label KILLPARAMS:

This section is called when it is discovered that a procedure body is a code statement. The allocation of the data space to hold the parameters and the link is cancelled, the parameter identifier records removed from DECLIST, and LINK and LABDECLIST (which hold records of 'LABEL' parameters) cleared.

Label LABL:

This section is called following the symbols 'LABEL'(Identifier) and after a jump instruction in code. It creates an Arithmetic Operand record of type integer, with the marker LBM set, to denote that the direct address contains a pointer to a label record.

Label LABSK:

This section is called after the occurrence of a subscripted identifier in a label context, to check that the identifier is the name of a switch. After the check, its type is changed to integer to allow its use in expressions by the use of 'LABEL'.

Label LABTRACE:

This section sets 'BIT'[0]TRACE to the value of ACCUMULATOR. It is called in the case of a 'NO' 'LABEL' 'TRACE' directive, in which case ACCUMULATOR will be zero; and also in the case of a 'LABEL' 'TRACE' directive, in which case the value of ACCUMULATOR will be 1.

Label LASTDIM:

This section is called to process the last dimension of multi-dimensional arrays. It uses DODIM to set up an Iliffe vector in Special Data, the number of entries of which is given by the current value of ARRAYS, which is the product of the number of arrays and the sizes of all previous dimensions. This Iliffe vector is required to be relocated with respect to the starting address of the Data area.

Label LHSBITS:

This section is called where it is required to assign an expression to a partword field of a variable. A check is made to ensure that the variable is not itself a partword table field, and is not of type 'FLOATING'. If this is discovered, an error message is output. (In the first case the assignment is inefficient and ambiguous, and in the second case could give rise to unnormalised numbers. In the first case the required sub-field should be specified as a separate field, and in the second the overlay facility may be used, if this action must be carried out.)

The type of the variable is changed to the integer type set up in IDTYPE, and the required field specification inserted in its PARTWORD field.

Label LHSPROC:

This section is called where a statement commences with an identifier which is not followed by a colon (label) or a becomes symbol or subscript (assignment). It checks that the identifier is the identifier of a procedure (of any type). If not, a warning message is output, and the PARAMSPEC field set to point to the appropriate dummy parameter specification record at the base of the stack, in order to avoid a subsequent syntax failure.

SCALEFIRM is cleared in case the procedure is a 'VALUE' 'PROCEDURE'. Although the result is not required, in this case it will be evaluated to type floating for the sake of consistency.

Label LIBADD:

This section is called in 'LIBRARY' declarations and communicators, after the integer constant specifying the Library reference number for the preceding identifier has been processed. This number is masked to ensure that it does not overflow into the tag field, which is set to 5. This address is inserted into the direct address field of the identifier record being formed at the base of the stack.

Label LOOKUPD:

This section creates an Arithmetic Operand record for a variable, and ensures that its address is placed in the DIRADD field. This section is called to obtain an address to be used either as an overlay base, or as an address part of a code instruction.

Label MIDDIM:

This section is called to process all the dimensions except the first and last of three or more dimensional arrays. It uses DODIM to set up an Iliffe vector in Special Data, the number of entries of which is given by ARRAYS. This Iliffe vector is required to be relocated with respect to the starting address of the Special Data area.

Label MINUSSUB:

This section is called, in a subscript expression, after each Term preceded by a - symbol. This section calls SUBTERM, which performs subscript arithmetic and optimisation.

Label MOREFOR:

This section is called after the occurrence of each comma separating elements in a for list. The value of FORSTATE gives the type of the preceding element(s).

If FORSTATE is zero, this occurring if the preceding element consists simply of an expression, and is the first element; an unconditional jump instruction is output, its address recorded in RTA, and FORSTATE is set to 2. Otherwise, if FORSTATE is less than four, the controlled statement is to be treated as a procedure, and a call is made of SETRT.

A call is made to FORCOM to output a call of the controlled statement, and other associated instructions. This is followed by calls of REVCHAIN and SETDUE to set the SKIPCHAIN, which is used when the previous element is exhausted, to the current program address. The current program address is recorded in BJA, for use if the following element is a while element, and ZEROACCS is called. ASSFUN is reset to its initial value of function 10, and DATASTART reset to its starting value, making allowance only for the index of the control variable. If the control variable is subscripted, and the subscript has been evaluated and stored in an anonymous location, the "hole-marker" flag in its indirect address will be set.

Label MPY:

This section is called to output instructions for the multiplication of two operands. As the operation may have the operands in either order, an attempt is made to have them in the order which would generate the least number of instructions. If the left hand operand is a constant the order is reversed, because the operand will not require to be unpacked, and may be an integer constant which is a power of two. Otherwise SWOPOPT is called. An accumulator is then chosen, using GOODACC.

The type and scale to which the multiplication is carried out depends only on the types and scales of the operands, and not on the context. It is probably better, with fixed point operands which are held left justified, to multiply first and then rescale if required.

If either operand is of type floating, then the operation is carried out in floating point arithmetic, by means of a call of FLOATOP, with a parameter of 1.

If the right hand operand is an integer constant which is a power of two, the multiplication will not be performed using a multiply instruction. If the left hand operand is of type integer, this will be performed by shifting, which will be combined with any unpacking shifts required. The number of places of shift will be added to the number of significant bits of the operand if this is non zero. If the left hand operand is of type fixed, the multiplication is carried out by adjusting the scale field of its operand record. In this case the number of significant bits in the record remains unaltered.

Label MPY/

If both operands are of type integer, the multiplication will be carried out as an integer multiplication, delivering an integer result. This is unfortunately rather inefficient if overflow is to be correctly detected. If the number of significant bits of both operands is given, the number of significant bits of the result is given by the sum of their significant bits, with allowance being made for the sign bit, subject to the restrictions on the word length. Otherwise, the number of significant bits of the result is unspecified.

If one operand is of type integer, and the other of type fixed, the integer operand is converted to the equivalent fixed scale. This is performed by rescaling, if the operand is a constant, or by shifting if the operand is not. In this case the number of significant bits of the integer operand must be specified.

If both operands are, or have been converted to, type fixed, the scale of the result is calculated from their scales and numbers of significant bits. The number of significant bits of the result is taken to be the smaller number of the two operands, and the scale calculated has its number of integer bits equal to the sum of the integer bits of the two operands. As this may exceed the range allowed by the implementation, a check is made, and a warning message output if required.

After the type and scale of the result has been calculated, the actual instructions for the multiplication are then output. The left hand operand is copied into the chosen accumulator, and if the right hand operand is required to be unpacked, or is marked as being a location, this is copied into another accumulator. The instruction is then output to perform the multiplication. If the type of the result is integer, three instructions are output to shift the result back into the accumulator. This section returns to the syntax analyser via BEHEADEXIT, which removes the top Arithmetic Operand record from the stack.

Label MSK:

This section is called to apply the 'MASK' operation to the two top Arithmetic Operands on the main stack. This is performed by means of a call of OPERATE, with function 15 as the parameter. As the function is reversible, i.e.

A 'MASK' B = B 'MASK' A

for all values of A and B, a call is first made on SWOPOPT to reverse the order of the operands, if this would result in less code being generated.

Label MULTICALL:

This section is called after each call of a multiple call of a procedure, except the final call. It resets the STACKPOINTER, to delete the previous set of parameters, and resets the PSP pointer to the parameter specification record.

If the procedure is typed, the result of the previous call will be used as the first (explicit) parameter of the next; and the type of this parameter must exactly be the same type as the type of the procedure, which must have more than one parameter. If the procedure is not a 'VALUE' procedure, the record for this parameter is set up by a single call of MAKEPARAM, with a parameter of zero. In the case of a 'VALUE' procedure, three calls are made to MAKEPARAM with parameters of 1, 0, and 1. This sets up three parameters which are, the type required for the result, the result of the previous call, and its type. Note that when compiling a call on a 'VALUE' procedure, a marker is set in PNP by SETUPPROC.

Label NEGNUM:

This section negates the variable NUMBER. It is called, in cases where a signed constant not occurring in an expression is preceded by a minus sign, after the number has been assembled.

Label NEQ:

This section is called to apply the 'DIFFER' operation to the two top Arithmetic Operands on the main stack. This is performed by means of a call of SWOPOPT, for optimisation purposes, followed by a call of OPERATE, with function 16 as the parameter.

Label NEWNAME:

This section is called after the occurrence of an identifier in a declaration. It creates an Identifier Specification Record on the stack. A check is made to ensure that this is the only occurrence of the identifier at this block level, and that it is not the (illegal) seventh parameter of a procedure. The record is then placed on the stack by ONSTACK, the relevant information being held in the "declaration breeding ground" at the base of the main stack. If the identifier is an indirectly addressed parameter of a procedure, the direct and indirect addresses in the record are exchanged, and the location marker deleted.

If the identifier is a parameter, a call is made to NEXTPARAM, to increment the Parameter Specification record for the procedure; and if the parameter is a 'LABEL', a label record is created by LOOKUPLAB, its reference being temporarily stored in the INDADD field of the parameter identifier record.

Label NEXTPSET:

This section is called between sets of parameters in procedure declarations, between parameters in parameter specifications, and after the context parameter in declarations of 'VALUE' procedures. This section clears the type and other fields of IDTYPE, leaving only the parameter accumulator field PAC, and the sign bit marker, unchanged. This allows the type of the following parameter to be built up in the usual manner.

Label NISACC:

This section is called where the operand of an instruction in code is an accumulator. An Arithmetic Operand record is set up by NONAME, and the accumulator number (with AMARK added) is inserted in the DIRADD field.

Label NISMOD:

This section is called where the operand of an instruction in code contains a modifier. After checking that the accumulator specified is the number of a modifier, this number (with AMARK added) is stored in the INDADD field of the Arithmetic Operand record of the instruction.

Label NOAFTER:

This section is called after the integer specifying the number of bits after the binary point has been processed. This occurs in 'FIXED' numbertypes, and also in table fields. The scale is calculated and inserted into the PVL field of IDTYPE, the TYP field being set to the value for 'FIXED' variables (1). The scale (or P value + 32) is given by:

Total bits - Number after point + 31

a check is made to ensure that this lies within the allowable range for this implementation. (0 to 63)

Label NOBITS:

This section is called after the Integer specifying the number of bits has been processed. In the case of normal declarations this specifies the number of significant bits required to hold the number (although a whole word will be used), and in the case of table fields and 'BITS' operation this specifies the number of bits in the partword. After checking that the number of bits is within the allowable range, the number is inserted into the SGB field of IDTYPE.

Label NOSIG:

This section is used in the formation of a partword descriptor held in BITSPEC. It is called after the occurrence of the integer specifying the number of bits required by a table field and also by a 'BITS' operation. The number of bits is subtracted from the number of bits in the computer word, and stored in BITSPEC. No check is made at this stage. The number held in BITSPEC will be subsequently used by FIELDPOSN.

Label OFFSTEXPR:

This section is called at the end of an untyped bracketed expression, and is also entered at the end of a typed bracketed expression. It removes one stack level for expressions, resetting the five variables starting with EXPRCHAIN. The top Arithmetic Operand record is moved down, and a call made on SETLHRH to reset the LH and RH pointers. STACKPOINTER is updated, and a call made on ACCUPDATE to update ACCS.

Label ONEBIT:

This section is called where the number of bits specified in a 'BITS' operation has been omitted, and is assumed to be one. The action of this section is identical to the actions carried out by NOBITS,NOSIG and PARTINT where NUMBER is one. BITSPEC is set to 23, the SGB field of IDTYPE set to one, and the PVL field set to integer scale.

Label ONEDIM:

This section is called to process the sole dimension of single dimensional arrays. ADDRARRAY is called to insert the addresses of the Data space allocated to the arrays, in the identifier records on the stack. The number of records to be processed is given in ARRAYS.

Label ORACT:

This section is called following the symbol 'OR' in a Condition. It outputs the test instruction, reversing the test, with the address part chained on the DUECHAIN; and sets the SKIPCHAIN. REVCHAIN is called (twice) to ensure that the procedures OUTCJ and SETDUE operate on the required chains.

Label ORF:

This section is called to apply the 'UNION' operation to the two top Arithmetic Operands on the main stack. This is performed by means of a call of SWOPOPT, for optimisation purposes, followed by a call of OPERATE, with function 17 as the parameter.

Label OUTCODE:

This section is called at the end of each instruction in code to output the completed instruction. The accumulator number is held in ACCUMULATOR, and the function number in FUNCTION. The top (RH) Arithmetic Operand record on the stack contains the information required for the N (address) and M (modifier) fields. The instruction is output by INST after the LCM field of the record has been cleared, as this may have been set by LABL or STRINGEX. This section returns to the syntax analyser via BEHEADEXIT which removes the top Arithmetic Operand record from the stack.

Label OUTPRESET:

This section is called after each numeric value or void in a Presetlist. The constant is rescaled, if required, by SCALECON, and output by OUTPRESETD.

Label OVEROFF:

This section is called at the end of an overlay declaration, and clears the OVERLAY flag.

Label OVERON:

This section is called after the symbol 'WITH', in an overlay declaration. The address of the start of the area to be overlayed is held in an Arithmetic Operand on the top of the stack, and is obtained using TOSADD, which deletes the operand. This address is held in OVERBASE and is inserted into the required location at the base of the stack, and the OVERLAY flag set.

Label OVRTEST:

This section is called following the symbol 'OVERFLOW' in a condition. FUNCTION is set to 24, the overflow test function; ACCUMULATOR being previously set, depending on the presence (0), or absence (1), of the symbol 'NO'.

Label PARAMTAB:

This section is called before, and after, the specification of the fields of a table parameter. It stores, and later restores, the current values of IDTYPE and DIRADD[0]. The information held in these (parameter accumulator, parameter address) is required for any subsequent parameters, and would be destroyed by the field specifications if not preserved. The variables IDTYPE2 and PARAMPTR2 are used for this purpose. (See EXCPSPEC)

Label PARTINT:

This section is called after the specification of a partword integer. It inserts the scale for integer (P=23) into the PVL field of IDTYPE.

Label PLUSSUB:

This section is called, in a subscript expression, after each Term preceded by a + symbol, and also after the first Term if unsigned. This section calls SUBTERM, which performs subscript arithmetic and optimisation.

Label PRESETSTRING:

This section is called after each quoted string in a Presetlist. The string is output, destined for the Special Constants area, by SPECSTRING, and the address of the string output, destined for the Data area and relocated, by OUTPRESETD.

As it is allowed to assign (the addresses of) strings to variables, the extension has been made to allow this assignment to be preset.

Label PROCENTRY:

This section is called when the entry point of a non-code procedure is reached.

The local procedure SCAN applies its procedure parameter PROC to each parameter identifier record in turn (if any), using the pointer P, which is non-local to tbe procedures.

The procedure LABPARAM is designed to be a parameter of SCAN. If the parameter record is that of a label parameter, an indirect jump instruction, with the Data address of the parameter as its address, is output, and a label with the same identifier as the parameter, set to this instruction.

The other procedure, DUMPPARAM, is also designed to be a parameter of SCAN. This procedure outputs an instruction to store a parameter, of the procedure being compiled, in the Data location allocated for this parameter. The number of the accumulator is obtained from the parameter identifier record, as is the address. No action is taken if the accumulator number is zero, as in this case the identifier record will be that of a field of a table parameter.

PROCENTRY starts with a call of SCAN, with LABPARAM as a parameter. This outputs one instruction for each label parameter, and creates a label identifier record. This enables (conditional) jumps to be made to the label parameter in the normal manner, using the STATUS optimisations. It is expected that each label parameter will be used at least once in the procedure body, and most probably in a conditional jump. (Error exits etc.) A call is then made on ENTERPROC, which causes the loader to set the contents of the location containing the entry point to the current program address. An instruction is then output to store the link; this being followed by a call of SCAN with DUMPPARAM as a parameter, to output the instructions to store the parameters. If procedure trace is required, a call on L4, with the procedure identifier string as a parameter, is then output.

Label PROCTRACE:

This section sets 'BIT'[1]TRACE to the value of ACCUMULATOR. It is called in the case of a 'NO' 'PROCEDURE' 'TRACE' directive, in which case ACCUMULATOR will be zero; and also in the case of a 'PROCEDURE' 'TRACE' directive, in which case the value of ACCUMULATOR will be 1.

Label RAISE:

This section is called to apply the exponentiation operation (↑) to its operands. If the right hand operand is not of type integer, the operation is carried out in floating point arithmetic by means of a call of FLOATOP with a parameter of 5. If the right hand operand is not the integer constant 2 or the left hand operand is of type floating, the operation is carried out by (fast) floating point arithmetic, by means of a call of FLOATOP with a parameter of 6.

The remaining case, squaring a fixed point number, is carried out by fixed point multiplication. The left hand Arithmetic Operand record is duplicated, replacing the right hand record (of the constant 2); if the operand exists only in an accumulator, an instruction is output to dump a copy of this operand into a temporary workspace. The section MPY is then entered.

Label RELADD:

This section is called where the operand of a jump instruction in code is a relative address, using 'SELF' or *. An Arithmetic Operand record is created by NONAME, and its commentary set to "(SELF)". The relative address is obtained by adding the current program address PTA, and the displacement held in NUMBER. This is performed using ADDADD, to avoid the possibility of overflow changing the tag field. The result is stored in the DIRADD field of the record.

Label RELATION:

This section is called after the second of the two expressions in a Comparison have been processed. It calculates the jump instruction required, which is stored in FUNCTION; and ensures that the result is in an accumulator, whose number is stored in ACCUMULATOR.

Where the test involves a single variable, its address, where possible, is used to update the accumulator record ACCS. If this variable is a partword, it will be unpacked without shifting where possible.

Label RHSBITS:

This section is called where a 'BITS' operation is used in an expression, after the expression to which the operation is to be applied has been processed. If the expression is an address, or is a partword field, or a constant; an instruction is output to copy it into an accumulator. This will provide redundant instructions, but will enable the operation to be carried out in an unambiguous manner. The partword specification and type information, which has been packed into BITSPEC by BITSIN is unpacked and inserted into the TYPEBITS and PARTWORD fields of the Arithmetic Operand. The actual bits operation being later carried out by PICK.

Label SCALETERM:

This section is called after each Term of an expression. If SCALEFIRM is set, the scale required for the expression has been defined, and the action of this section is confined to changing the type of the Terms to or from 'FLOATING' as required. If the required type is floating, a call is made on FLOATIT to ensure that the Term is of, or will be converted to, type floating. Otherwise, if the type of the Term is floating, a call to L21 is output, to convert the Term to the required fixed point scale.

On the other hand, if SCALEFIRM is not set, the scale of the expression is determined Term by Term. EXPSCALE is initially set to zero, and as each Term is processed, it is progressively modified according to the types and scales of the Terms. The type of the expression progressively goes through the sequence 'INTEGER' to 'FIXED' to 'FLOATING' according to the "strongest" type found. A comparison between the current value of EXPSCALE and the SCALE of all Terms except the first, is made by the use of SCALESTEST to switch to the appropriate label. In the case of the first Term, EXPSCALE is taken as the SCALE of the Term. Using the notation given in the description of SCALETEST, the action taken for all Terms after the first is summarised below.

Label SCALETERM/

1 Io,Io & Io,Is & Is,Io

The scale is taken to be integer, with unspecified significance.

2 Is,Is

The scale is taken to be integer, with the larger number of significant bits.

3 & 4 Is,Fx & Fx,Is

The scale of the integer type is converted to the equivalent fixed scale (provided that the number of significant bits is known) and the scale determined as in section 5.

Note however, that when an integer is treated as a fixed, it is regarded as having an infinite number of (zero) fraction bits.

5 Fx,Fx

A new fixed scale is calculated, which has the larger number of integer bits, and the smaller number of fraction bits, of the two scales. If this would exceed the wordlength, fraction bits are dropped.

6 Fl,Any & Any,Fl

In this case the type is taken as floating.

Label SETANS:

This section is called after the occurrence of the 'ANSWER' symbol. It checks that an answer statement is allowed, and sets EXPSCALE and SCALEFIRM for the following expression. If the procedure is a 'VALUE' procedure, these are both cleared, thus allowing the type to be determined by the following expression, it being assumed that the context parameter is used to dynamically determine this. On the other hand, if the procedure is typed, the expression following the 'ANSWER' symbol is evaluated to the type of the procedure. In this case EXPSCALE is set to the type and scale of the procedure, and the flag SCALEFIRM is set. An answer statement is not allowed:

A. In an untyped procedure

B. In a 'CODE' procedure

C. Outside a procedure.

Note that this section starts with the declaration of DUMMYBLOCK, which makes this section a block, and allows the use of FAULT as a local label.

Label SETLAB:

This section is called after the occurrence of an identifier setting a label. STATUSCODE is called, to complete any deferred action associated with STATUS, to set it to zero, and to set any chain held by DUECHAIN. SETLABEL with LOOKUPLAB as a parameter, looks up and creates a label record if required, inserts the current program address into the record, and sets any chains of previous usage of the label.

Label SETLB:

This section is called after the integer constant giving the lower bound to a dimension of an array has been processed. The number is stored negatively in OFFSET.

Label SETLEVEL:

This section sets the value of the variable LEVEL, to the value of NUMBER. This section is called after a 'LEVEL' directive, to set the diagnostic level of the compiler output. At present, as implemented there are four diagnostic levels, 0 to 3, the default value being 1. The effects are as follows:

Level	Diagnostics
0	None
1	Page titles, labels, procedures and for statements printed at compile time.
2	As above, plus output of label, procedure, and for statement information to the loader.
3	As above, plus detailed commentary on each instruction output to the loader.

Label SETLIBSEG:

This section is called at the start of a Library procedure segment. The segment header is output by STARTSEG, and the housekeeping procedure, PROCSTACK, called. The DUECHAIN is set to the procedure address (containing the reference number), and marked for the convenience of the loader. The first Data location is allocated for the link, subsequent ones being used for the parameters and local workspace. TRACE is cleared, as procedur4s requiring trace should not be placed in the Library.

Label SETLIBVAR:

This section is called for the (trivial) compilation of Library variables. A check is first made to ensure that the preset value of this variable is greater than (in integer convention) -8372225 ($-2^{23}+2^{14}-2^{0}$). This allows all floating point numbers, and most fixed and integers.

A tape is then produced giving the following information to the loader:

a) The Library Number of the variable

b) Its Preset value

c) Its Identifier string

d) A directive tag (4) and checksum.

This information is repeated on the printer.

Label SETNEZ:

This section is called where the comparator and second expression have been omitted in a comparison. In this case it is assumed that the comparison is against zero, with a zero value counting as "false", and a non-zero value counting as "true". This allows conditions such as:

```
'IF' 'BIT'[9]A 'THEN' . . .
```

RELOP is set to zero, which is the value it would have received if the comparator < > had been read.

Label SETNO:

This section sets the value of the variable ACCUMULATOR, to 0. It is called in cases where the symbol 'NO' is optional, and is present. Examples of such cases are:

```
'IF' 'NO' 'OVERFLOW' 'THEN' . .
'NO' 'PROCEDURE' 'TRACE';
```

Label SETPARAMS:

This section is called when it is discovered that a procedure body is not a code statement. This confirms the allocation of the data space required to hold the parameters and the link, and records the position on the stack of the identifier record of the last parameter (if any) in PARAMDECS.

Label SETPREF:

This section is called to select a suitable accumulator for the evaluation of bracketed expressions. An accumulator number is supplied by FINDPREF, and stored in PREFACC. If the preferred accumulator of the previous expression level is free, this will be chosen.

Label SETSHIFT:

This section is called after the occurrence of a shift operator in an expression, to prepare for the optimisation of the expression giving the number of shifts required. The required shift function is copied into BITSHIFT, and an anonymous Arithmetic Operand record created. The section SETUPSUB is then entered.

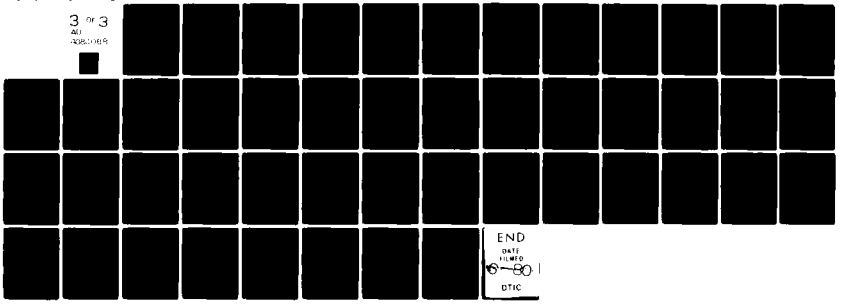
Label SETTEN:

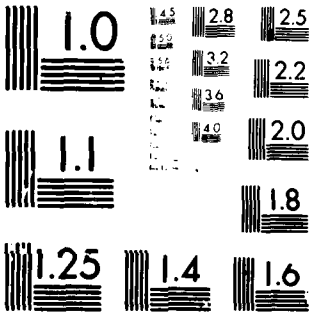
This section copies the constant held in the variable NUMBER into TOPTEN. It is called where two integer constants may occur in close succession. The value of the first is held in TOPTEN until the second has been processed. This section is called in the case of packed address constants in 'SPECIAL' 'ARRAY's, and in the case of the base and displacement of 'EXTERNAL' specifications.

Label SETTEST:

This section sets the value of the variable TEST, to 1. It is called in cases where the symbol 'TEST' is optional, and is present. This allows test compilations to be performed. The effect of setting TEST is to inhibit the paper tape output on the punch. TEST is reset to zero at the end of each segment.

AD-A084 068 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
THE CORAL 66 COMPILER FOR FERRANTI ARGUS 500 COMPUTER.(U)
JUN 78 B GORMAN
UNCLASSIFIED RSRE-TN-799 DRIC-BR-67199 NL





Label SETTEST/

'TEST' may also be used where a 'COMMON' communicator has been previously compiled, and it is required to set up the compiler for the compilation of a segment referring to it, without producing an output tape.

Label SETUB:

This section is called after the integer constant giving the upper bound to a dimension of an array has been processed. The size of this dimension is calculated in NUMBER, and if less than 1, an error message is output, and NUMBER set to one.

Label SETUPSUB:

This section is called after the occurrence of the starting symbol of a subscript ([), and is also entered from SETSHIFT.

A check is first made that the Arithmetic Operand on the top of the stack may be legally subscripted, if not an error message is output. A new expression level is created by a call of STACKEXPR, and this level set to indicate that an integer expression is mandatory. A suitable modifier is chosen as the preferred accumulator.

Label SETYES:

This section sets the value of the variable ACCUMULATOR, to 1. It is called in cases where the symbol 'NO' is optional, and has been omitted. Examples of such cases are:

```
'IF' ('NO') 'OVERFLOW' 'THEN' . .
('NO') 'PROCEDURE' 'TRACE';
```

This section is also called at the start of the declaration of an untyped procedure. Until the symbol following 'PROCEDURE' has been processed, it is not possible to determine whether a procedure declaration or a trace directive is to be processed. The call in these circumstances is harmless. As a consequence of this, the section TYPEPROC is also called.

Label SIMPLEASS:

This section is called where an assignment takes the form of:

Variable Becomes ($\pm$) Term

to optimise the cases of:

Variable Becomes ($\pm$) Variable.

If the variable on the right is the same as the variable on the left, and is unsigned or preceded by a + symbol, ASSFUN will have been set to function 12. In this case ASSFUN is cleared, and no instructions will be output. This case may occur as follows:

```
'FOR' A ← A 'WHILE' . . .
```

If the Term on the right is preceded by a - symbol, and is not the same as the variable on the left, and neither is of type floating; function 11 will be used for the assignment where appropriate, otherwise a call is made of UNARYMINUS, and function 10 used. This section is not called if assignment trace is required.

Label SKIPDTA:

This section is called following the Becomes symbol in a data declaration. Provided presetting is allowed, a directive is output, if required, to the loader, to skip the Data transfer address forwards, over unpreset areas of data.

Label SPECCON:

This section is called after the occurrence of an integer constant or identifier, preceded by a / symbol, in the presetlist of a 'SPECIAL' 'ARRAY'. An Arithmetic Operand record will have been created on the top of the stack, this is deleted and its direct address field obtained by TOSADD. This address is packed and output by OUT1014CS.

Label SPECLAB:

This section is called after the occurrence of an identifier, preceded by a : symbol, in the presetlist of a 'SPECIAL' 'ARRAY'. This identifier is treated as a label identifier, and its treatment differs in program and common segments. This section is also used to process entries in switch lists.

In a program segment, the label record is accessed using LOOKUPLAB, and this reference to the label chained up. It should be noted that the label cannot have been set at this block level before its use.

In a common segment the label may or may not have previously been explicitly specified, or used in a special array. A scan is made of the identifier records of the common segment. If a record does not already exist, a new one is created, as if the label were being explicitly specified. Otherwise, after checking that the record is that of a common label, this reference to the label is chained up to the previous reference.

In either case, the chain address of the label is output using OUT1014CS.

Label SPECNUM:

This section is called after the occurrence of a numeric constant in the presetlist of a 'SPECIAL' 'ARRAY'. The constant is rescaled to the scale held in IDTYPE and output by OUT24CS. It should be noted that, although the scale of a 'SPECIAL' 'ARRAY' is integer, preset numeric constants may be stored to other scales if specified.

Label SPECONE:

This section is called after each identifier of a label, switch, or procedure, declared in a common segment. A location in special data, whose address will have been obtained using ADDRSPEC, is reserved, and set to an initial value of zero. The loader will subsequently insert the requisite address into this location.

Label SPECREL:

This section is called after the occurrence of a relative reference ('SELF' or *) in the presetlist of a 'SPECIAL' 'ARRAY'. The displacement, held in NUMBER, is added to the current Special Data address STA, and output by OUT10104CS, which inserts an integer into the top ten bits if required.

Label STARTDEC:

This section is called before all declarations, to initialise the "declaration breeding ground" at the base of the main stack.

Label STARTFOR:

This section is called after the occurrence of the symbol 'FOR'. A call is made of STATUSCHECK to complete any deferred actions, and to ensure that the statement can be entered. A call is made of BEGLABBLOCK to ensure that the controlled statement cannot be entered by means of a jump to a label from a point outside. The values of the working variables used for the compilation of for statements:

FORCHAIN,FORSTATE,CV,COPYSKIP,DATAADD,BJA,RTA,RTL, and CCC

are stored by a call of ONSTACK, in case the for statement occurs within another. COPYSKIP is set to the current value of SKIPCHAIN, which is then cleared, as this is required for use in Conditions following 'WHILE' and also for the jump made when a for element is exhausted. The current value of DATASTART is recorded in DATAADD, so that it may be reset at the end of the for level. This value is also recorded in RTL.

FORSTATE, which will be used to determine the complexity of the list of for elements, and the type of each, is set to its initial value of zero.

Label STEPUNT:

This section is called after the expressions following the 'STEP' and 'UNTIL' symbols in a 'FOR' statement have been processed. CCC is used to detect the case where all three expressions are constants, and is calculated by a logical and ('MASK') of the constant markers of the three expressions. The type of the expression is changed, if required, to the type of the control variable. If the expression is a constant, this is performed by means of a call of SCALENUM, otherwise a call is made of PICK. One is added (twice per Forelement) to FORSTATE, to indicate that the for element is of the form step-until. This section returns to the syntax analyser via FORSTORE, which stores the expression in an anonymous workspace if it has been evaluated.

Label STORE:

This section is called at the end of an assignment statement, to output the instructions for the assignment. This is performed by means of a call of STOREAWAY with the assignment trace flag, stored in TRACE, as the parameter.

Label STOREZERO:

This section is called when it is discovered that an assignment takes the form of:

Variable Becomes Zero.

If the variable is a Wordreference, a marker is set in ASSFUN to indicate that accumulator 0 is to be used with the assignment function (10). If the variable is a partword, then the optimisation will be carried out using FFLAG in STOREAWAY. This causes the field to be set to zero by masking the rest of the word. This section is not called if assignment trace is required.

Label STRINGEX:

This section is called after the occurrence of a quoted string as an expression, or in code, to create an Arithmetic Operand of type integer for the string. The string is output by SPECSTRING, and its address placed in the DIRADD of the record.

Label SUB:

This section is called to output the instructions for the subtraction of one Term from another. This is performed by means of a call of ADDSUB with function 03 as the parameter.

Label SUBA:

This section is called after the first Term of the expression on the right hand side of an assignment symbol; if the Term is preceded by a - symbol. If the Term has already been evaluated in an accumulator, a call is made of UNARYMINUS to output an instruction to negate it. Otherwise, if the term is a variable, and is the same variable as the one on the left of the assignment symbol, and is not a partword or of type floating, ASSFUN is set to function 13, so that a "negate-store-and-add-to-store" assignment will be used. A mark is then set in ASSFUN, to indicate to SIMPLEASS or SELOPTA that the term was preceded by a minus sign. This section is not called if assignment trace is required.

Label SUBCOMMA:

This section is called after the occurrence of a comma in a subscript. A local pointer(LH) to the Arithmetic operand of the variable being subscripted, is calculated. If the indirect address of this variable is zero, this only occurring after the first comma where the previous expression consisted of an integer constant, the direct address is transferred to the indirect address of the record, and the direct address cleared. Otherwise, a suitable modifier is chosen, and an instruction is output, to copy the contents of the address so far calculated, into it. (This instruction may itself be modified.) The modifier number and accumulator marker (AMARK) are placed in the indirect address field of the Arithmetic Operand record, and the direct address field cleared. Finally a call is made on ACCUPDATE to update the accumulator record, ACCS.

Label TABADD:

This section is called before the fields of a table are specified. It stores the direct and indirect addresses of the start of the table in TABLED and TABLEI, for subsequent use by FIELDDISP. If the direct address of the table is null (Table parameters) it has a marker set to ensure that all table fields are treated in a consistent manner by LOOKUPD.

Label TABLESIZE:

This section is called after the occurrence of the Integer constant giving the number of locations required for a table. Provided that the table is not being overlayed, the number of locations is added to DATAMAX.

Label THENEX:

This section is called following the occurrence of the symbol 'THEN' in a conditional expression. The final test instruction is output by OUTCJ, and any chain resulting from the use of 'OR' set by SETDUE.

Label TYPEARRAY:

This section is called to set the AYM of IDTYPE, to denote that the following identifiers are identifiers of an array. This occurs in the case of arrays declared within a program or common segment, in the case of arrays specified in external and absolute communicators, and also for every table field.

Label TYPEFLOAT:

This section is called, following the symbol 'FLOATING', to insert the type number for floating point (2), into the TYP field of IDTYPE.

Label TYPEIARRAY:

This section is called in the case of array and table parameters. It sets the AYM and LCM bits of IDTYPE to denote that the location of an array or table is required to be passed as a parameter.

Label TYPEINT:

This section is called after the occurrence of the symbol 'INTEGER' in the specification of an integer of unspecified significance. It is also used in 'TABLE' and 'SPECIAL' 'ARRAY' declarations, both of which are treated as integer arrays.

This section inserts the scale for integer (P=23) into the PVL field of IDTYPE.

Label TYPEISWITCH:

This section is called following the symbol 'SWITCH' in common communicators, and in procedure declarations and specifications. It sets the TYP field of IDTYPE to the type number for labels (3), and sets the AYM and LCM flags. The LCM flag indicates that the switch is to be indirectly addressed.

Label TYPELAB:

This section is called, following the symbol 'LABEL' in declarations, to insert the type number for labels (3), into the TYP field of IDTYPE.

Label TYPELOC:

This section is called following the symbol 'LOCATION' in procedure specifications and declarations. It sets the marker LCM in IDTYPE, to denote that a location is required as the actual parameter. The marker LM2 is also set, this being used for non-standard parameters.

Label TYPEPROC:

This section is called, preceding the symbol 'PROCEDURE', to insert the type number for untyped procedures (4), into the TYP field of IDTYPE.

Label TYPESPEC:

This section is called in the case of non-standard (untyped) parameters, in a procedure declaration or specification. In this case the symbols 'VALUE' and 'LOCATION' are not followed by a Numbertype. The SPB field of IDTYPE is set to 1, to denote the first parameter of the pair (the second being the type/scale), and the PVL field is set to the default scale. (Integer)

Label TYPESWITCH:

This section is called following the symbol 'SWITCH' in switch declarations, and 'EXTERNAL' and 'ABSOLUTE' switch specifications. It inserts the type number for labels (3) into the TYP field, and sets the array marker AYM, of IDTYPE. Thus a switch is treated as an array of labels.

Label TYPETPROC:

This section is called, following a Numbertype, and preceding the symbol 'PROCEDURE', to convert the TYP field of IDTYPE, from a type number for a variable, to a type number for a typed procedure.

Number Type	Variable type number	Procedure type number
'INTEGER'	0	5
'FIXED'	1	6
'FLOATING'	2	7

Label TYPEVPROC:

This section is called between the symbols 'VALUE' and 'PROCEDURE' to insert the type information into the TYP (5) and SPB (3) fields of IDTYPE, for a variable-type procedure.

Label TYPEXPR:

This section is called following a Numbertype preceding a bracketed expression. In this case the expression is required to be evaluated to the type and scale stated, even if the context demands a differing type or scale. (In this case the value will be rescaled to the type demanded by the context after the evaluation of the bracketed expression.) A new expression level is created by a call of STACKEXPR, and the EXPSCALE of this new level set to the required scale. SCALEFIRM is set to indicate that this scale is mandatory. The section SETPREF is then entered.

Label UNSFIELD:

This section is called to complete the partword descriptor BITSPEC, and to make appropriate adjustments to IDTYPE, in the case of 'UNSIGNED' table fields, and 'BITS' operations. A check is made to ensure that a 24 bit unsigned field is not being specified; and the UNS marker of BITSPEC is then set. One is added to the SGB field of IDTYPE, and if the scale is 'FIXED', one is added to the PVL (scale). This makes allowance for the sign bit, which although not present, is allowed for when unpacking and aligning the field.

Label VARCHECK:

This section is called to check that an identifier, used as a variable, is an arithmetic variable; i.e. is of type integer, fixed, or floating. If it is not, a warning message is output, and its type changed to integer to allow compilation to continue. This section returns to the syntax analyser via AYMCHECK to ensure that a subscript is not required, or has already been supplied.

Label WHILEL:

This section is called after the Condition following a 'WHILE' symbol in a 'FOR' statement has been processed. The final jump-if-false is output by a call of OUTCJ, and the destination of any jump-if-true instructions, set by means of a call of SETDUE. One is added to FORSTATE to indicate that the for element is of the while form.

Label ZEROARRAYS:

This section is called at the start of a set of arrays having common bounds. The location, ARRAYS, which keeps count of the number of arrays is cleared. TOPTEN is also cleared so that the top ten bits of any addressing vectors generated will be clear.

Label ZEROCOMP:

This section is called where the expression on the right of a comparator is zero. In this case two expressions do not require comparison. If the comparator is > or <=, the left hand expression requires negation, as the instruction set of the machine does not contain the appropriate jump instructions.

Label ZERONUM:

This section clears the variable NUMBER. It is called in cases where a constant is optional, and has been omitted. In this case its value is assumed to be zero.

List of Errors

Page:	ARITHMETIC
Procedure:	FLOATOP
Line:	3
After:	ACCUPDATE(LH);
Insert:	TYPEBITS[LH]←//10000;
Page:	ASSIGNMENT AND FOR
Procedure:	STOREAWAY
Line:	4
After:	PARTWORD[RH]=0
Insert:	'AND' PARTWORD[LH]=0
Line:	20
Replace:	24-SHIFT)'MASK' FORMMASK(RH)
By:	24+SHIFT)'MASK' FORMMASK(LH)
Page:	TABLES AND SPECIAL
Label:	SPECLAB
Line:	3
Before:	'COTO' NEWNAME
Insert:	OUT1014CS(ADD);
Page:	CALLS AND CODE
Label:	CALLPROC
Line:	6
Replace:	PICK(LH,A,TYPEBITS[LH],0)
By:	PICK(LH,A,STB[LH],0)

```
'PAGE' DEFINITIONS;
```

1

```
'EXTERNAL'('INT' EXT1/1,EXT2/2,EXT3/3);

( TYPES )

'DEFINE' INTEGER "0";
'DEFINE' FIXED "1";
'DEFINE' FLOATING "2";
'DEFINE' LABEL "3";
'DEFINE' PROCEDURE "4";
'DEFINE' INTPROC "5";( FIXPROC = 6, FLOATPROC = 7 );

( CONSTANTS )

'DEFINE' MARK "#40000000";
'DEFINE' AMARK "#77740000";

( FORMULAE )

'DEFINE' CYCLE(AAA,BBB) "((AAA)'SLC'(BBB))";
'DEFINE' CHECKSUM(AAA,BBB) "AAA+(AA'SLC'1 'DIFFER' (BBB)";

( AREAS )

'DEFINE' MCSTART "'LOC'(EXT1)";
'DEFINE' STACKSTART "'LOC'(EXT2)";
'DEFINE' STACKFINISH "'LOC'(EXT3)";
```

```
'PAGE' STACK;
```

2

```
COMPILER20

'BEGIN' 'TABLE' STACK[39][

CHAIN 'INT' 0;( CHAIN OF STACKED ITEMS )
SPIFL 'INT' 0;( STRING FOR COMMENTARY )

TYPEBITS 'INT' 1;( DETAILS OF ITEM ON STACK )
CON 'UNS'(1) 1'BIT'0; ( CONSTANT MARKER )
AYM 'UNS'(1) 1'BIT'1; ( ARRAY MARKER )
LCM 'UNS'(1) 1'BIT'2; ( LOCATION MARKER )
PAC 'UNS'(3) 1'BIT'3; ( PARAMETER ACCUMULATOR )
LM2 'UNS'(1) 1'BIT'6; ( COPY OF LCM )
LBM 'UNS'(1) 1'BIT'7; ( 'LAB' MARKER )
SPB 'UNS'(2) 1'BIT'8; ( VARIABLE TYPE )
[illegible] 'UNS'(14)1'BIT'10;( SCALE AND TYPE )
TYP 'UNS'(3) 1'BIT'10;( TYPE )
SGB 'UNS'(5) 1'BIT'13;( NO OF SIGNIFICANT BITS )
PVL 'UNS'(6) 1'BIT'18;( SCALE FACTOR )

PARAMSPEC 'INT' 2;( POINTER TO PROC PARAMETER SPECS )

PARTWORD 'INT' 2;( PART WORD SPECIFICATION )
UNS 'UNS'(1) 2'BIT'0; ( MARKER FOR UNSIGNED FIELD )
LSS 'UNS'(5) 2'BIT'14;( SHIFTS TO ALIGN AT BOTTOM )
MSS 'UNS'(5) 2'BIT'19;( SHIFTS TO ALIGN AT TOP )

DIRADD 'INT' 3;( DIRECT ADDRESS )
ACD 'UNS'(1) 3'BIT'0; ( MARKER FOR DIRECT ADDRESS IS ACC )
HMD 'UNS'(1) 3'BIT'1; ( MARKER FOR HOLE )
PTF 'UNS'(1) 3'BIT'2; ( PARAMETRIC TABLE FIELD )
TGD 'UNS'(3) 3'BIT'4; ( TAG OF DIRECT ADDRESS )
DAD 'UNS'(14)3'BIT'10;( DIRECT DISPLACEMENT )

INDADD 'INT' 4;( INDIRECT ADDRESS )
ACI 'UNS'(1) 4'BIT'0; ( MARKER FOR INDIRECT ADDRESS IN ACC )
HMI 'UNS'(1) 4'BIT'1; ( MARKER FOR HOLE )
TGI 'UNS'(3) 4'BIT'6; ( TAG OF INDIRECT ADDRESS )
IAD 'UNS'(14)4'BIT'10;( INDIRECT DISPLACEMENT )

STRING 'INT' 5;( IDENTIFIER HELD AS STRING )
WDS 'UNS'(4) 5'BIT'0; ( NUMBER OF WORDS-1 USED BY STRING )
CHS 'UNS'(6) 5'BIT'0; ( NUMBER OF CHARS IN STRING )

'COMMENT' LABEL DECLARATION LISTS;

PCH 'INT' 1; ( PROGRAM CHAIN )
DCH 'INT' 2; ( DATA CHAIN )
LLS 'UNS'(4) 3'BIT'0; ( LENGTH OF LABEL STRING (-1) )
LABID 'INT' 3;( LABEL IDENTIFIER )

]+0,0,0,0,0,0,0,0,0,0,0,0,070,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,#07010000,-1;

'OVERLAY' STACK[0] 'WITH' 'INT' DECLIST,IDTYPE/BITSPEC,ARRAYS;

'OVERLAY' STACK[5] 'WITH' 'INT' NAME;'OVERLAY' STACK[29] 'WITH' 'INT' MCID;
```

```
'PAGE' GLOBAL;                                                                  3

'INT' 'ARRAY' ACCS[0:7];'INTEGER' OUTDEV=#106;

'INTEGER' T,T1,ACCUMULATOR,FUNCTION,NACC,NUMBER,NUMBERSCALE,STACKPOINTER,LH,RH,STATUS,TEST,AGGFUN;

( TRANSFER ADDRESSES )

'INTEGER' PTA,DTA,STA;

( DECLARATIONS )

'INTEGER' TOPTEN,OFFSET,COMMON,TABLED,TABLET,OVERLAY,OVERBASE,IDTYPE2,PARAMPTR2,TSUM,CSUM;

( BLOCKS )

'INTEGER' BLOCKCHAIN,LEVEL,TRACE,LOCALLIMIT,DATASTART,DATAPTR,DATAMAX,PRESETOK;

( LABEL BLOCKS )

'INTEGER' LABCHAIN,LABDECLIST,LABSTACKPTR;

( PROCEDURE DECS )

'INTEGER' PROCCHAIN,PROCPTR,PARAMPTR,PARAMDECS,LINK,PROCSTRING,EXITCH,DUECHAIN;

( CONDITIONALS )

'INTEGER' IFCHAIN,SKIPCHAIN,COPYDUE,RELOP;

( EXPRESSIONS AND PROCEDURE CALLS )

'INTEGER' EXPRCHAIN,PREFACC,EXPSCALE,SCALEFIRM,BITSHIFT,PNP,PSP,FPP;

( FOR STATEMENTS )

'INTEGER' FORCHAIN,FORSTATE,CV,COPYSKIP,DATAADD,BJA,RTA,RTL,CCC;

( MACROS )

'INTEGER' MCCHAIN,T2,MCSOURCE,MCBODY,MCLIST,MCTOP,MCFLAG;
```

```
'PAGE' NASTY CODE;                                                              4

'INTEGER' INCTR;'INT' 'ARRAY' INBUFF[0:63];

'INTEGER' 'PROCEDURE' READTAPE;'CODE' 'BEGIN'

R:0 JBS R;
  7 LDX [5];
  7 AND #177;
  7 SUB #12;
  7 JZE S;
  7 SUB #26;
  7 JLT R;
  7 SUB #100;
  7 JLT S;
  7 SUB #37;
  7 JZE R;
  7 SUB 1;
S:7 ADD #40;
  7 JGE *2;
  7 NEG #20;
  7 ADD #40;
  1 LDX INCTR;
  7 STO INBUFF[B1];
  1 ADD 1;
  1 AND #77;
  1 STO INCTR;
  0 JCS 04
'END' READTAPE;

'PROC' DUMMY;'CODE' 'BEGIN' 0JCS04 'END' DUMMY;

'PROC' SWOP('LOC' 'INT' X,Y);'CODE' 'BEGIN' 1LDXB7;2LDXB6;7LDX[B1];7EXC[B2];7STO[B1];0JCS04 'END' SWOP;

'PROC' MOVE('VAL' 'INT' N;'LOC' 'INT' FROM,TO);'CODE' 'BEGIN' 1LDXB6;2LDXB5;1ADDB7;2ADDB7;
                    1SUB1;2SUB1;7SUB1;6LDX[B1];6STO[B2];7JNZ*-5;0JCS04 'END' MOVE;

'INT' 'PROC' NEXTCHAR('LOC' 'INT' Z);'CODE' 'BEGIN' 2LDXB7;1LDX[B2];7LDX[B1];1JLT*2;7SRL12;1SLC1;1JLT*2;7SRL6;
                    7AND#77;1SLC1;1ADD1;1SLC22;1STO[B2];0JCS04 'END' NEXTCHAR;

'INT' 'PROC' TESTSTRING('VAL' 'INT' S1,S2);'CODE' 'BEGIN' 1LJXB7;2LDXB6;6LDXSTACK[B1];6SRL20;
          7LDXSTACK[B1];7NEGSTACK[B2];7JNZ*5;1ADD1;2ADD1;6SUB1;6JGE*-6;0JCS04 'END' TESTSTRING;
```

```
'PAGE' OUTPUT 1;


'PROCEDURE' OUTCHAR('VAL' 'INT' CHAR);'CODE' 'BEGIN'

7 SUB #100;
6 NLX #163;     ( #400 - CR )
7 JGE OUT;      ( CR LF REQUIRED )
7 ADD #40;
7 JGE *2;
7 NEG #20;
7 ADD #100;
6 LDX #322664626;
6 STO [2];
6 LDX #226;
3 LDX B7;
6 SLL 03;
6 AND #200;
6 ADD B7;

OUT 1

1 LDX OUTDEV;
7 LDX [B1];
7 JLT *-1;
6 OUT [B1];
6 JLT *2;
0 JCS 84;
6 LDX #12;
0 JZE OUT

'END' OUTCHAR;

'PROCEDURE' TEXT('VAL' 'INT' STRING);
        'BEGIN' 'INTEGER' I;'FOR' I:=1,1;NEXTCHAR(STRING) 'DO' OUTCHAR(NEXTCHAR(STRING)); 'END' TEXT;

'PROCEDURE' NEWLINE;OUTCHAR(#100);

'PROCEDURE' TEXTLINE('VAL' 'INT' S);'BEGIN' TEXT(S);NEWLINE 'END' TEXTLINE;

'PROC' HALT('VAL' 'INT' STRING);'BEGIN' TEXT("HALTED - ");TEXTLINE(STRING);
        L;'IF' [4]<0 'THEN' 'GOTO' L;M;'IF' [4]>=0 'THEN' 'GOTO' M 'END' HALT;

'PROC' OCTLOOP('VAL' 'INT' WORD,DIGITS);'BEGIN' WORD:=CYCLE(WORD,(8-DIGITS)*3);
   'FOR' DIGITS:=DIGITS-1 'WHILE' DIGITS>=0 'DO' 'BEGIN' WORD:=CYCLE(WORD,3);OUTCHAR(WORD 'MASK' 7) 'END' 'END' OCTLOOP;

'PROC' PRINTADD('VAL' 'INT' ADD);'BEGIN' 'SPECIAL' 'ARRAY' TAG=SA,SP,SD,SS,SC,SL,SE,SW;
     OUTCHAR(TAG['BITS'[3,6]ADD]);OCTLOOP(ADD 'MASK' #37777,5);TEXT(": ")
'END' PRINTADD;

'PROCEDURE' PRINTBUFF;'BEGIN' 'INT' I,C;TEXT("? . . ");;
     'FOR' I:=INCTR,(I+1)'MASK'#77 'WHILE' I<>INCTR 'DO'
          'BEGIN' C:=INBUFF[I];'IF' C>=0 'THEN' OUTCHAR(C);'IF' C=#100 'THEN' OUTCHAR($7) 'END' FOR I;
     TEXTLINE("***")
'END' PRINTBUFF;

'PROCEDURE' GIVEUP('VAL' 'INT' S);'BEGIN' PRINTBUFF;TEXT("FAILED : ");TEXTLINE(S);'GOTO' STARTUP 'END' GIVEUP;
```



```
'PAGE' OUTPUT 2;


'PROCEDURE' OUT5('VAL' 'INT' TAG,WORD);'IF' TEST=0 'THEN' 'BEGIN' 'INT' I;OUTDEV:=#107;'FOR' I:=4;-1;0 'DO'
   'BEGIN' OUTCHAR(TAG);CHECKSUM(TSUM,TAG);WORD:=CYCLE(WORD,6);TAG:=WORD 'MASK' #77 'END' LOOP;
OUTDEV:=#106 'END' OUT5;

'PROCEDURE' OUTCONT('VAL' 'INT' WORD);OUT5(0,WORD);

'PROCEDURE' OUT24('LOC' 'INT' TA;'VAL' 'INT' WORD,TAG);'BEGIN' TA:=TA+1;
        OUT5('BITS'[3,6]TA+8+'BITS'[3,6]TAG,WORD) 'END' OUT24;

'PROCEDURE' OUT1014('LOC' 'INT' TA;'VAL' 'INT' TEN,ADD);
        OUT24(TA,CYCLE(TEN,14)'MASK' #77740000 + ADD 'MASK' #37777,ADD);

'PROCEDURE' OUT1014CS('VAL' 'INT' ADD);OUT1014(STA,TOPTEN,ADD);

'PROCEDURE' OUT24CS('VAL' 'INT' CONST);OUT24(STA,CONST,0);

'PROCEDURE' PUNCHLOOP('VAL' 'INT' CHAR,TIMES);'CODE' 'BEGIN' SLDXTEST;SJNZOUT;
        SLDX[#107];SJLT*-1;7OUT[#107];6SUB1;6JNZ*-4;OUT:0JCS84 'END' PUNCHLOOP;

'PROCEDURE' THEAD;'BEGIN' PUNCHLOOP(0,100);TSUM:=0 'END' THEAD;

'PROCEDURE' TTAIL;'BEGIN' THEAD;OUT5(7,#03645760);PUNCHLOOP(#377,20);THEAD 'END' TTAIL;

'PROCEDURE' ENDSEG;'BEGIN' OUT5(39,TSUM);TEST:=0;TTAIL 'END' ENDSEG;

'PROCEDURE' STOPOP('VAL' 'INT' TA,TB);'BEGIN' TEXT(TA);TEXT(" : ");TEXT(TB);
   'IF' TEST=0 'THEN' 'BEGIN' TEST:=1;TEXTLINE(" ; COMPILATION INHIBITED") 'END' 'ELSE' NEWLINE;
    PRINTBUFF
'END' STOPOP;

'PROCEDURE' WARN('VAL' 'INT' TA,TB);'BEGIN' TEXT("WARNING : ");TEXT(TA);
   'IF' TB=0 'THEN' NEWLINE 'ELSE' 'BEGIN' TEXT(" : ");TEXTLINE(TB) 'END';
    PRINTBUFF
'END' WARN;
```

```
'PAGE' OUTPUT 5;
```



```
'PROCEDURE' DOSTRING('VAL' 'INT' STRING;'PROC' PROC('VAL' 'INT'));
      'FOR' STRING+STRING;1;STRING+'BITS'[4,0][STRING] 'DO' PROC([STRING]);

'INT' 'PROC' SPECSTRING;'BEGIN' 'INT'ANS;ANS+STA;DOSTRING('LOC'(NAME),OUT24CS);'ANS' ANS 'END' SPECSTRING;

'PROCEDURE' OUTI('VAL' 'INT' XFM,N,SPIEL);'BEGIN' 'INT' STRING;
      'IF' N<0 'THEN' 'BEGIN' SPIEL+'LOC'(STRING);N+N+#0005000O;STRING+N+#U3174000 'END' ACC
      'ELSE' 'IF' 'BIT'[1]N<>0 'THEN' SPIEL+"(TEMP)"
      OUT1014(PTA,XFM,N);'IF' LEVEL>=3 'THEN' 'BEGIN' DOSTRING(SPIEL,OUTCONT);OUT5(34,PTA-1) 'END'
'END' OUTI;

'PROCEDURE' OUTXFMN('VAL' 'INT' X,F,M,N,SPIEL);OUTI((X*32+F)*64+M,N,SPIEL);

'PROCEDURE' OUT27('VAL' 'INT' ADD,SPIEL);OUTI(#1134,ADD,SPIEL);

'PROCEDURE' CALLLIB('VAL' 'INT' LNO,SPIEL);OUT27(#00500000+LNO,SPIEL);

'PROCEDURE' LIBTRACE('VAL' 'INT' LNO,ADD,SPIEL);
      'BEGIN' OUTI(#1620,ADD,SPIEL);CALLLIB(LNO,"(TRACE)") 'END' LIBTRACE;

'PROCEDURE' TRACESTRING('VAL' 'INT' LNO,STRING);
      'BEGIN' LIBTRACE(LNO,STA,STRING);DOSTRING(STRING,OUT24CS) 'END' TRACESTRING;

'INT' 'PROC' OUTWCONST('VAL' 'INT' CONST);'BEGIN' OUTCONT(CONST);'ANSWER' #00700000 'END' OUTWCONST;

'INT' 'PROC' FORMMASK('VAL' 'INT' REF);
      'CODE' 'BEGIN' 1LDX07;1LDXPARTWORD(01];7LDX-1;7SRL01;1SRL5;7SRL01;7SLC01;0JCS04 'END' FORMMASK;

'PROCEDURE' MASKINST('VAL' 'INT' REF,ACC,SENSE);OUTXFMN(ACC,#15,0,OUTWCONST(FORMMASK(REF)'DIFFER'SENSE),"(MASK)");

'PROCEDURE' DIAG('VAL' 'INT' TA,TB);'IF' LEVEL<>0 'THEN' 'BEGIN'
    PRINTADD(PTA);TEXT(TA);TEXT(" ");TEXTLINE(TB);
    'IF' LEVEL>=2 'THEN' 'BEGIN' DOSTRING(TB,OUTCONT);OUT5(33,PTA) 'END'
'END' DIAG;
```

```
'PAGE' RESCALE;
```



```
'INT' 'PROC' RESCALE('VAL' 'INT' NUMBER,OLDSCALE,NEWSCALE);'CODE' 'BEGIN'

7 JZE EXIT       ;( NUMBER=0 )
6 AND #00014077  ;( MASK SCALES )
5 AND #00014077  ;
6 NEQ 05         ;( DIFFER )
A JZE EXIT       ;( TYPES/SCALES EQUAL )
6 NEQ 05         ;( RESTORE )
0 STD [2]        ;( CLEAR Q )
0 OVR *1         ;( CLEAR OVERFLOW )
7 SLV 03         ;( NORMALISE,03=32-SHIFTS LEFT )
7 SRA 1          ;
7 NEQ #40000000  ;( RESTORE SIGN )
3 SUB 31         ;( 03=TOTAL LEFT SHIFTS )
3 ADD 06         ;( 03=OLDSCALE(CORRECTED) )
6 LDX 03         ;
6 AND #77        ;( 06=OLD P VALUE +32 )
5 SSB 03         ;( 03=NEWSCALE-OLDSCALE )
3 JLT FAIL       ;( TYPE/SCALE MISSMATCH )
5 SLC 11         ;
5 JLT FLOAT      ;( FLOATING REQUIRED )
3 AND #777       ;
3 SUB 24         ;
3 JGE FAIL       ;( RESCALE>23 PLACES )
7 SRA 03*24      ;( RESCALE )
7 ADD [1]        ;( ROUND )

EXIT :

0 JCS 04         ;( EXIT )

FLOAT;

0 OVR *1         ;( CLEAR OVERFLOW )
7 ADD #40        ;( ROUND )
1 OVR *3         ;( IF NO OVERFLOW )
7 SRL 1          ;
6 ADD 1          ;( CORRECT SCALE )
7 AND #77777700  ;( MASK M,S, 18 BITS )
7 STN [0]        ;( TEST FOR -1,0 )
1 OVR *3         ;( NOT -1,0 )
7 SRA 1          ;
6 AND 1          ;
7 ADD 06         ;( PACK )
6 AND #77777700  ;
6 JZE EXIT       ;( EXPONENT NOT OUT OF RANGE )

FAIL :

7 LDC "NUMBER"   ;
6 LDC "CANNOT BE RESCALED";
0 JCS STOPOP

'END' RESCALE;

'INT' 'PROC' SCALECON;'BEGIN' NUMBER+RESCALE(NUMBER,NUMBERSCALE,IDTYPE);'ANSWER' NUMBER 'END' SCALECON;
```

```
'PAGE' ACCUMULATORS;
```



```
'PROCEDURE' ZEROACCS;'BEGIN' 'INT' A;'FOR' A←1;1;7 'DO' ACCS[A]←0 'END' ZEROACCS;

'PROCEDURE' DUMPACC('VAL' 'INT' A);'BEGIN' OUTXFNN(A,#10,0,@ATAPTR,"(DUMP)");STACK[ACCS[A]]←@ATAPTR←#20000000;
      ACCS[A]←0;@ATAPTR←@ATAPTR+1;'IF' @ATAMAX<@ATAPTR 'THEN' @ATAMAX←@ATAPTR
'END' DUMPACC;

'PROCEDURE' DUMPACCS;'BEGIN' 'INT' A;'FOR' A←1;1;7 'DO' 'IF' ACCS[A]<0 'THEN' DUMPACC(A) 'END' DUMPACCS;

'INT' 'PROC' FINDACC('VAL' 'INT' A);'BEGIN' 'INT' ANS,TEST,MIN;MIN←-1;'FOR' A←A;-1;1 'DO'
    'BEGIN' TEST←('IF' ACCS[A]=0 'THEN' 0 'ELSE' ACCS[A] 'DIFFER' #20000000)-MIN;
      'IF' [3]<>0 'THEN' 'BEGIN' MIN←MIN+TEST;ANS←A 'END' IF
    'END' SCAN OF ACCS;
    'IF' MIN<0 'THEN' DUMPACC(ANS) 'ELSE' ACCS[ANS]←0;
'ANSWER' ANS 'END' FINDACC;

'INT' 'PROC' FINDPREF;'ANSWER' 'IF' ACCS[PREFACC]=0 'THEN' PREFACC 'ELSE' FINDACC(7);

'INT' 'PROC' FINDOUT('VAL' 'INT' ADD);'IF' ADD<0 'OR' ADD=0 'THEN' 'ANSWER' ADD 'MASK' 7 'ELSE'
      'BEGIN' 'INT' A;'FOR' A←1;1;7 'DO' 'IF' ACCS[A]=ADD 'THEN' 'ANSWER' A;'ANSWER' 0 'END' FINDOUT;

'PROCEDURE' ACCUPDATE('VAL' 'INT' REF);'IF' TYPEBITS[REF]>=0 'THEN'
    'IF' DIRADD[REF]<0 'THEN' ACCS[DIRADD[REF]]←REF+3←MARK 'ELSE'
    'IF' INDADD[REF]<0 'THEN' ACCS[INDADD[REF]]←REF+4←MARK;

'PROCEDURE' INACC('VAL' 'INT' ACC,REF);'BEGIN' DIRADD[REF]←ACC←AMARK;INDADD[REF]←0;ACCUPDATE(REF) 'END' INACC;

'PROCEDURE' TOPACC('VAL' 'INT' ACC);INACC(ACC,RH);

'INT' 'PROC' ACCOF('VAL' 'INT' REF);'ANS' 'IF' TYPEBITS[REF]>=0 'AND' DIRADD[REF]<0 'THEN' DIRADD[REF]-AMAR. 'ELSE' 0;

'INT' 'PROC' LHACC;'ANSWER' ACCOF(LH);

'INT' 'PROC' RHACC;'ANSWER' ACCOF(RH);

'INT' 'PROC' COPYINACC('VAL' 'INT' REF);'IF' TYPEBITS[REF]<0 'OR' LCM[REF]<>0 'OR' INDADD[REF]<>0 'THEN' 'ANSWER' 0
      'ELSE' 'ANSWER' FINDOUT(DIRADD[REF]);

'INT' 'PROC' ISINACC('VAL' 'INT' REF);
      'BEGIN' 'INT' A;A←COPYINACC(REF);'IF' A<>0 'THEN' INACC(A,REF);'ANSWER' A 'END' ISINACC;

'PROCEDURE' PERM;
      'BEGIN' 'INT' I;'FOR' I←0;1;4 'DO' SWOP(STACK[LH+I],STACK[RH+I]);ACCUPDATE(RH);ACCUPDATE(LH) 'END' PERM;
```

```
'PAGE' STACK AND LABELS;
```



```
'PROCEDURE' STACKCHECK;'IF' STACKPOINTER>LABSTACKPTR 'THEN' GIVEUP("STACK COLLISION");

'INT' 'PROC' GRABSTACK('VAL' 'INT' N);
        'BEGIN' STACKPOINTER←STACKPOINTER+N;STACKCHECK;'ANSWER' STACKPOINTER-N 'END' GRABSTACK;

'PROCEDURE' ONSTACK('VAL' 'INT' N;'LOC' 'INT' START);
        'BEGIN' MOVE(N,START,STACK[STACKPOINTER]);START←GRABSTACK(N) 'END' ONSTACK;

'PROCEDURE' OFFSTACK('VAL' 'INT' N;'LOC' 'INT' START);
        'BEGIN' STACKPOINTER←START;MOVE(N,STACK[STACKPOINTER],START) 'END' OFFSTACK;

'PROCEDURE' SETCHAINTOPTA('VAL' 'INT' CHAIN);'IF' CHAIN<>0 'THEN' 'BEGIN' OUTS(17,CHAIN);ZEROACCS 'END' SETCH;

'PROCEDURE' JOINCHAINS('VAL' 'INT' CHAIN;'LOC' 'INT' MASTER);'IF' CHAIN<>0 'THEN' 'IF' MASTER=0 'THEN' MASTER←CHAIN
      'ELSE' 'BEGIN' OUTCONT(MASTER);OUTS('IF' MASTER<0 'THEN' 18 'ELSE' 20,CHAIN) 'END' JOINCHAINS;

'PROCEDURE' BEGLABBLOCK;'BEGIN' ONSTACK(3,LABCHAIN);LABDECLIST←0 'END' BEGLABBLOCK;

'PROCEDURE' ONLAB('VAL' 'INT' FROM);'BEGIN' 'INT' SIZE,NEWSP;
      SIZE←LLS[FROM]+4;NEWSP←LABSTACKPTR←SIZE;CHAIN[FROM]←0;MOVE(SIZE,STACK[FROM],STACK[NEWSP]);
      'IF' LABDECLIST=0 'THEN' LABDECLIST←NEWSP 'ELSE' CHAIN[LABSTACKPTR]←NEWSP;LABSTACKPTR←NEWSP
'END' ONLAB;

'PROCEDURE' ENDLABBLOCK;'BEGIN' 'INT' NEXT,DECS,OLDSP,T;NEXT←LABDECLIST;OFFSTACK(3,LABCHAIN);OLDSP←LABSTACKPTR;
   'FOR' DECS←NEXT 'WHILE' DECS<>0 'DO' 'BEGIN' NEXT←CHAIN[DECS];'IF' PCH[DECS]>=0 'THEN'
      'BEGIN' T←LABDECLIST;'FOR' T←T 'WHILE' T>=OLDSP 'DO' 'IF' TESTSTRING(T+3,DECS+3)=0 'THEN'
            'BEGIN' JOINCHAINS(PCH[DECS],PCH[T]);JOINCHAINS(DCH[DECS],DCH[T]);'GOTO' SKIP 'END' 'ELSE' T←CHAIN[T];
        ONLAB(DECS)
      'END' LABEL UNSET;
SKIP;'END' FOR DECS
'END' ENDLABBLOCK;

'INT' 'PROC' SCANLAB('VAL' 'INT' PTR);'BEGIN' 'INT' LAB;LAB←LABDECLIST;
      'FOR' LAB←LAB 'WHILE' LAB<>0 'AND' TESTSTRING(LAB+3,PTR+5)<>0 'DO' LAB←CHAIN[LAB];'ANSWER' LAB
'END' SCANLAB;

'INT' 'PROC' LOOKUPLAB;'BEGIN' 'INT' PTR;PTR←SCANLAB(0);'IF' PTR<>0 'THEN' 'ANSWER' PTR;
      ONLAB(2);PCH[LABSTACKPTR]←0;DCH[LABSTACKPTR]←0;STACKCHECK;'ANSWER' LABSTACKPTR
'END' LOOKUPLAB;

'INT' 'PROC' LOOKUPNAME('VAL' 'INT' LIMIT,REF);'BEGIN' 'INT' PTR;PTR←DECLIST;'FOR' PTR←PTR 'WHILE' PTR<>LIMIT 'DO'
      'IF' TESTSTRING(REF+5,PTR+5)=0 'THEN' 'ANSWER' PTR 'ELSE' PTR←CHAIN[PTR];
      'ANSWER' 0  'END' LOOKUPNAME;

'PROCEDURE' FIXLABEL('VAL' 'INT' LAB);'IF' PCH[LAB]<0 'THEN' STOPOP('LOC'(LABID[LAB]),"LABEL SET TWICE")
'ELSE' 'BEGIN' ZEROACCS;SETCHAINTOPTA(PCH[LAB]);SETCHAINTOPTA(DCH[LAB]);PCH[LAB]←PTA←MARK;DCH[LAB]←PTA←MARK;
      DIAG("LABEL",'LOC'(LABID[LAB]))
'END' FIXLABEL;

'PROCEDURE' SETLABEL('VAL' 'INT' LAB);
    'BEGIN' FIXLABEL(LAB);'IF' 'BIT'(0,TRACE)<>0 'THEN' TRACESTRING(3,'LOC'(LABID[LAB])) 'END' SETLABEL;

'INT' 'PROC' USELAB('VAL' 'INT' LAB);'BEGIN' 'INT' ANS;ANS←PCH[LAB];
      'IF' ANS<0 'THEN' ANS←ANS-MARK 'ELSE' PCH[LAB]←PTA;
'ANSWER' ANS 'END' USELAB;
```

```
'PAGE' CHAINS AND BLOCKS;
```



```
'PROC' SETDUE;'BEGIN' SETCHAINTOPTA(DUECHAIN);DUECHAIN←0 'END' SETDUE;

'PROC' REVCHAIN;SWOP(DUECHAIN,SKIPCHAIN);

'PROC' OUTCJ;'BEGIN' OUTXFMN(ACCUMULATOR,FUNCTION,0,SKIPCHAIN,"(TEST)");SKIPCHAIN←PTA-1 'END' OUTCJ;

'PROC' OUTUJ;'BEGIN' OUTI(#100,SKIPCHAIN,"(SKIP)");SKIPCHAIN←PTA-1 'END' OUTUJ;

'PROC' REVCJ;'IF' FUNCTION=#24 'THEN' ACCUMULATOR←ACCUMULATOR 'DIFFER' 1 'ELSE' FUNCTION←FUNCTION 'DIFFER' 1;

'PROC' DUEERR;'IF' DUECHAIN=0 'THEN' 'BEGIN' PRINTADD(PTA);STOPOP(0,"STATEMENT CANNOT BE ENTERED") 'END' DUEERR;

'PROC' ANSLINK;'IF' PROCSTRING=0 'THEN' 'BEGIN' OUT27(LINK,"(ANSWER)");EXITCH←1 'END'
        'ELSE' 'BEGIN' OUTI(#100,EXITCH,"(ANSWER)");EXITCH←PTA-1 'END' ANSLINK;

'PROC' STATUSTEST('PROC' PROC);'BEGIN' 'SWITCH' S←S0,S1,S2,S3,S4;'GOTO' S[STATUS+1];
     S4;OUTCJ;'GOTO' S0;
     S3;OUTUJ;'GOTO' S0;
     S2;ANSLINK;
     S1;PROC;
     S0;SETDUE;STATUS←0
'END' STATUSTEST;

'PROC' STATUSCHECK;STATUSTEST(DUEERR);

'PROC' STATUSCODE;'BEGIN' STATUSTEST(DUMMY);ZEROACCS 'END' STATUSCODE;

'INT' 'PROC' DIRINDADD('VAL' 'INT' REF);'ANSWER' 'IF' DIRADD[REF]<>0 'THEN' DIRADD[REF] 'ELSE' INDADD[REF];

'PROCEDURE' BEGINBLOCK;'BEGIN' BEGLABBLOCK;ONSTACK(8,BLOCKCHAIN);LOCALLIMIT←DECLIST;DATAMAX←DATASTART 'END' BEGINBLOCK;

'PROCEDURE' ENDBLOCK;'BEGIN' 'INT' MAX;MAX←DATAMAX;DECLIST←LOCALLIMIT;
     OFFSTACK(8,BLOCKCHAIN);'IF' DATAMAX<MAX 'THEN' DATAMAX←MAX;ENDLABBLOCK
'END' ENDBLOCK;

'PROCEDURE' FINISHSEG;'BEGIN' 'INT' DECS;'IF' LABDECLIST<>0 'THEN' 'BEGIN' TEXTLINE("UNSET LABELS");
     'FOR' LABDECLIST←LABDECLIST.CHAIN[LABDECLIST] 'WHILE' LABDECLIST<>0 'DO'
        'BEGIN' FIXLABEL(LABDECLIST);TRACESTRING(2,'LOC'(LABID[LABDECLIST])) 'END' FOR;
     LABSTACKPTR←STACKFINISH 'END' UNSETS;
     ENDSEG;THEAD;'FOR' DECS←PTA,DATAMAX,STA 'DO' 'BEGIN' OUTCONT(DECS);PRINTADD(DECS) 'END';
     TEXTLINE("END OF SEGMENT");OUTS(1,TSUM);TTAIL 'END' FINISHSEG;

'PROCEDURE' STARTSEG('VAL' 'INT' TYPE);'BEGIN' 'INT' STRING;STRING←'IF' TYPE=3 'THEN' "(COMMON)" 'ELSE' 'LOC'(NAME);
    THEAD;DOSTRING(STRING,OUTCONT);OUTS(60+TYPE,'IF' TYPE=2 'THEN' DIRADD[DECLIST] 'ELSE' CSUM);
    TEXT('IF' TYPE=1 'THEN' "PROGRAM" 'ELSE' 'IF' TYPE=2 'THEN' "LIBRARY" 'ELSE' "COMMON ");
    TEXT(" SEGMENT : ");TEXTLINE(STRING);
    PTA←#001000000;DTA←#002000000;DATAMAX←DTA;DATASTART←DATAMAX;STA←#003000000;ZEROACCS
'END' STARTSEG;
```

```
'PAGE' STACK AND PROCS;
```



```
'INT' 'PROC' ADDADD('VAL' 'INT' ADD,INC);'ANSWER' ADD 'MASK' #77740000 + (ADD+INC) 'MASK' #37777;

'PROCEDURE' SETLHRH;'BEGIN' RH←STACKPOINTER-5;LH←RH←5 'END' SETLHRH;

'PROCEDURE' BEHEAD;'BEGIN' 'IF' TYPEBITS[RH]>=0 'THEN'
     'IF' DIRADD[RH]<0 'THEN' ACCS[DIRADD[RH]]←0 'ELSE' 'IF' INDADD[RH]<0 'THEN' ACCS[INDADD[RH]]←0;
        STACKPOINTER←RH;SETLHRH 'END' BEHEAD;

'INT' 'PROC' TOSADD;'BEGIN' 'INT' A;A←DIRADD[RH];BEHEAD;'ANSWER' A 'END' TOSADD;

'PROCEDURE' GRABVAR;'BEGIN' GRABSTACK(5);SETLHRH;PARTWORD[RH]←0;DIRADD[RH]←0;INDADD[RH]←0 'END' GRABVAR;

'PROCEDURE' NONAME;'BEGIN' GRABVAR;SPIEL[RH]←"(ANON)";TYPEBITS[RH]←#20000067 'END' NONAME;

'PROCEDURE' LOOKUP;'BEGIN' 'INTEGER' PTR;GRABVAR;PTR←LOOKUPNAME(0,0);
     'IF' PTR<>0 'THEN' 'BEGIN' SPIEL[RH]←'LOC'(STRING[PTR]);MOVE(4,TYPEBITS[PTR],TYPEBITS[RH]) 'END'
     'ELSE' 'BEGIN' TYPEBITS[RH]←#00000067;SPIEL[RH]←"(UNDECLARED)";TEXT('LOC'(NAME));TEXTLINE(" : UNDECLARED") 'END'
'END' LOOKUP;

'PROCEDURE' XSPARAMS('VAL' 'INT' PTR);
    'BEGIN' STOPOP('LOC'(STRING[PTR]),"HAS TOO MANY PARAMETERS");IDTYPE←IDTYPE-MARK 'END' XSPARAMS;

'PROCEDURE' NEXTPARAM;'IF' IDTYPE<0 'THEN' XSPARAMS(0) 'ELSE' 'IF' PAC[0]<>0 'THEN' 'BEGIN'
    STACK[PARAMPTR]←IDTYPE;PARAMPTR←PARAMPTR+1;STACK[PARAMPTR]←-1;
    PAC[0]←PAC[0]-1;'IF' PAC[0]=4 'THEN' PAC[0]←3 'ELSE' 'IF' PAC[0]=0 'THEN' IDTYPE←IDTYPE←MARK;
    'IF' SPB[0]=1 'THEN' 'BEGIN' SPB[0]←2;LCM[0]←0 'END' 'ELSE'
    'IF' SPB[0]=2 'THEN' 'BEGIN' SPB[0]←1;LCM[0]←LM2[0] 'END'
'END' NEXTPARAM;

'PROCEDURE' LOCACT;'BEGIN' PARTWORD[RH]←0;'IF' DIRADD[RH]<>0 'THEN' TYPEBITS[RH]←#10000067 'ELSE'
     'BEGIN' DIRADD[RH]←INDADD[RH];INDADD[RH]←0;TYPEBITS[RH]←#00000067;ACCUPDATE(RH) 'END' 'END' LOCACT;

'PROCEDURE' OUTPRESETD('VAL' 'INT' N,T);'IF' PRFSETOK=0 'OR' DTA>=DATAMAX 'THEN' STOPOP('LOC'(NAME),"ILLEGAL PRESET")
     'ELSE' OUT24(DTA,N,T);

'PROCEDURE' EXCPSPEC;'BEGIN' SWOP(IDTYPE,IDTYPE2);SWOP(PARAMPTR,PARAMPTR2) 'END' EXCPSPEC;

'PROCEDURE' MAKEPSPEC;'BEGIN' PARAMPTR2←GRABSTACK(7);PARAMSPEC[DECLIST]←PARAMPTR2←MARK;STACK[PARAMPTR2]←-1;
        IDTYPE2←'IF' SPB[0]<>0 'THEN' #07100067 'ELSE' #07000000 'END' MAKEPSPEC;

'PROCEDURE' FINISHPSPEC;'BEGIN' STACKPOINTER←PARAMPTR2+1;PARAMSPEC[DECLIST]←PARAMSPEC[DECLIST]←MARK 'END' FINISHPSPEC;

'PROCEDURE' PROCSTACK;'BEGIN' STATUS←1;MAKEPSPEC;ONSTACK(8,PROCCHAIN);BEGINBLOCK;EXCPSPEC;PROCPTR←DECLIST;
        LOCALLIMIT←CHAIN[DECLIST];EXITCH←0;PROCSTRING←0 'END' PROCSTACK;

'PROCEDURE' ENTERPROC;
    'BEGIN' SETDUE;STATUS←0;'IF' LEVEL<>0 'THEN' PRINTADD(DIRADD[PROCPTR]);DIAG("PROC ",'LOC'(STRING[PROCPTR]))
'END' ENTERPROC;
```

```
'PAGE' ARRAYS AND EXPRESSIONS;
```



```
'INT' 'PROC' ARRAYBASE;'ANSWER' 'IF' OVERLAY<>0 'THEN' OVERBASE 'ELSE' DATAMAX;

'PROCEDURE' DODIM('VAL' 'INT' START,INC;'PROC' ACT('VAL' 'INT');'BEGIN' 'INTEGER' I;START←ADDADD(START,OFFSET);
      'FOR' I←1;I;ARRAYS 'DO' 'BEGIN' ACT(START);START←ADDADD(START,INC) 'END' FOR I;
      ARRAYS←ARRAYS←NUMBER;'GOTO' EXIT
'END' DODIM;

'PROCEDURE' ADDRARRAY('VAL' 'INT' START);'BEGIN' 'INTEGER' PTR;
   'PROCEDURE' MINE('VAL' 'INT' ADD);'BEGIN' DIRADD[PTR]←ADD;PTR←CHAIN[PTR] 'END' MINE;
     PTR←DECLIST;DODIM(START←(ARRAYS-1)←NUMBER,-NUMBER,MINE)
'END' ADDRARRAY;

'PROCEDURE' STACKEXPR;'BEGIN' ONSTACK(5,EXPRCHAIN);EXPSCALE←0;SCALEFIRM←0 'END' STACKEXPR;

'PROCEDURE' KILLEXPR;'BEGIN' OFFSTACK(5,EXPRCHAIN);SETLHRH 'END' KILLEXPR;

'INT' 'PROC' ISMOD('VAL' 'INT' REF);'BEGIN' 'INT' MOD,ADD;ADD←INDADD[REF];'IF' ADD=0 'THEN' 'ANSWER' 0;
   MOD←FINDOUT(ADD);'IF' MOD=0 'OR' MOD>=4 'THEN'
       'BEGIN' MOD←FINDACC(3);OUTXFMN(MOD,00,0,ADD,"(INDEX)");'IF' ADD<0 'THEN' ACCS[ADD]←0 'END' LOADMOD;
   ACCS[MOD]←'IF' 'BITS'[2,0]ADD<>0 'THEN' 0 'ELSE' ADD;'ANSWER' MOD 'END' ISMOD;

'PROCEDURE' INST('VAL' 'INT' REF,ACC,FUN);OUTXFMN(ACC,FUN←('IF' LCM[REF]<>0 'THEN' 4 'ELSE' 0),ISMOD(REF),
   'IF' TYPEBITS[REF]<0 'THEN' OUTWCONST(DIRADD[REF]) 'ELSE' 'IF' LBM[REF]<>0 'THEN' USELAB(DIRADD[REF])
                  'ELSE' DIRADD[REF],SPIEL[REF]);

'PROCEDURE' SCALENUM('VAL' 'INT' REF,SCALE);
   'BEGIN' DIRADD[REF]←RESCALE(DIRADD[REF],TYPEBITS[REF],SCALE);TYPEBITS[REF]←SCALE 'UNION' MARK 'END' SCALENUM;

'PROCEDURE' PICK('VAL' 'INT' REF,ACC,SCALE,NEG);'BEGIN' 'INT' FUN,SHIFT;

'IF' TYPEBITS[REF]<0 'THEN' SCALENUM(REF,SCALE);SHIFT←'BITS'[6,18]SCALE-'BITS'[6,18]TYPEBITS[REF];

'IF' COPYINACC(REF)<>ACC
 'THEN' 'BEGIN' 'IF' PARTWORD[REF]<>0 'OR' NEG=0 'THEN' FUN←0 'ELSE' 'BEGIN' FUN←1;NEG←0 'END';INST(REF,ACC,FUN) 'END';

'IF' PARTWORD[REF]<>0 'THEN' 'BEGIN' MASKINST(REF,ACC,0);
   'IF' PARTWORD[REF]<0 'THEN' SHIFT←SHIFT←('IF' TYP[REF]=INTEGER 'THEN' LSS[REF] 'ELSE' 1-MSS[REF])
   'ELSE' 'BEGIN' 'IF' MSS[REF]<>0 'THEN' OUTXFMN(ACC,#33,0,MSS[REF],"(ALIGN)");
                  'IF' TYP[REF]=INTEGER 'THEN' SHIFT←SHIFT←MSS[REF]←LSS[REF] 'END'
'END' PART WORD UNPACKING;

'IF' SHIFT<>0 'THEN' 'BEGIN' 'IF' SHIFT<0 'THEN' 'BEGIN' SHIFT←-SHIFT;FUN←#31 'END' COULD DIAG POSSIBLE OVR
   'ELSE' FUN←'IF' PARTWORD[REF]<0 'THEN' #32 'ELSE' #30;
   'IF' SHIFT>=24 'THEN' WARN("EXCESS SHIFT","TO RESCALE");OUTXFMN(ACC,FUN,0,SHIFT,"(SCALE)");
   'IF' 'BITS'[3,10]SCALE=INTEGER 'AND' TYP[REF]=FIXED 'AND' SGN[REF]>AVL[REF]-#37 'THEN' OUTXFMN(ACC,02,0,1,"(ROUND)")

'END' SHIFTING;

ACCS[ACCOF(REF)]←0;TYPEBITS[REF]←SCALE;PARTWORD[REF]←0;INACC(ACC,REF);'IF' NEG<>0 'THEN' INST(REF,0,#13)

'END' PICK UNPACK AND RESCALE;
```

```
'PAGE' SUBSCRIPTS AND SCALING;
```



```
'INT' 'PROC' GOODONE('VAL' 'INT' REF,LIM);'BEGIN' 'INT' ANS;
   ANS←FINDOUT('IF' TYPEBITS[REF]<0 'THEN' AMARK 'ELSE' 'IF' INDADD[REF]<>0 'THEN' INDADD[REF]
       'ELSE' 'IF' LCM[REF]=0 'AND' DIRADD[REF]<>0 'THEN' DIRADD[REF] 'ELSE' AMARK);
'ANS' 'IF' ANS<>0 'AND' ANS<=LIM 'THEN' ANS 'ELSE' 'IF' ACCS[PREFACC]=0 'AND' PREFACC<=LIM 'THEN'
       PREFACC 'ELSE' FINDACC(LIM) 'END' GOODONE;

'PROCEDURE' SUBTERM('VAL' 'INT' NEG);'BEGIN' 'INT' A,LH;LH←EXPRCHAIN-5;( LH IS LOCAL POINTER )
   'IF' TYPEBITS[RH]<0 'THEN'
       'BEGIN' SCALENUM(RH,#1767);DIRADD[LH]←ADDADD(DIRADD[LH],'IF' NEG=0 'THEN' DIRADD[RH] 'ELSE' -DIRADD[RH]) 'END'
   'ELSE' 'IF' INDADD[LH]=0 'THEN'
       'IF' NEG<>0 'OR' TYP[RH]<>INTEGER 'OR' PARTWORD[RH]<>0 'OR' LBM[RH]<>0 'OR' LCM[RH]<>0 'AND' TGD[LH]<>0
           'OR' LCM[RH]=0 'AND' INDADD[RH]<>0 'THEN'
               'BEGIN' A←GOODONE(RH,3);PICK(RH,A,#1767,NEG);INDADD[LH]←A←AMARK 'END'
       'ELSE' 'IF' LCM[RH]<>0 'THEN' 'BEGIN' DIRADD[LH]←ADDADD(DIRADD[RH],DIRADD[LH]);INDADD[LH]←INDADD[RH] 'END' LOC
       'ELSE' INDADD[LH]←DIRADD[RH]
   'ELSE' 'BEGIN'
       'IF' PARTWORD[RH]<>0 'OR' TYP[RH]<>INTEGER 'THEN'
           'BEGIN' A←GOODONE(RH,'IF' NEG<>0 'OR' INDADD[LH]<AMARK+4 'THEN' 7 'ELSE' 3);PICK(RH,A,#1767,0) 'END';
       'IF' NEG=0 'AND' DIRADD[RH]<AMARK+4 'THEN' SWOP(DIRADD[RH],INDADD[LH]);
       A←ISMOD(LH);'IF' INDADD[LH]<>INDADD[RH] 'THEN' ACCS[A]←STACKPOINTER+MARK;
       INST(RH,A,NEG+02);INDADD[LH]←A+AMARK
   'END' ADD SUBTRACT;
BENFAD;ACCUPDATE(LH);'GOTO' EXIT 'END' SUBTERM;

'INT' 'PROC' GOODACC('VAL' 'INT' REF);'ANSWER' 'IF' ISINACC(REF)<>0 'THEN' ACCOF(REF) 'ELSE' FINDPREF;

'PROCEDURE' GOODPICK('VAL' 'INT' REF,NEG);PICK(REF,GOODACC(REF),STB[REF],NEG);

'PROCEDURE' SWOPOPT;
   'IF' RHACC=PREFACC 'OR' LHACC=0 'AND' RHACC<>0 'OR' ISINACC(LH)=0 'AND' ISINACC(RH)<>0 'OR' TYPEBITS[LH]<0
       'THEN' PERM;

'PROCEDURE' OPERATE('VAL' 'INT' FUN);'BEGIN' GOODPICK(LH,0);
   'IF' PARTWORD[RH]<>0 'OR' LCM[RH]<>0 'THEN' GOODPICK(RH,0);
   INST(RH,LHACC,FUN);TYPEBITS[LH]←#67;'GOTO' BENEADEXIT
'END' OPERATE;

'PROCEDURE' SCALETEST('VAL' 'INT' A,B;'SWITCH' S);'BEGIN' 'INT' T,V;
   T←(A 'MASK' #3700)+(B 'MASK' #3700);A←'BITS'[3,10]A;B←'BITS'[3,10]B;V←A 'UNION' B;
   'IF' V=0 'THEN' 'IF' T=0 'THEN' 'GOTO' S[1] 'ELSE' 'GOTO' S[2];
   'IF' V>#FLOATING 'THEN' 'GOTO' S[6];
   'IF' T=0 'THEN' 'BEGIN' WARN("OPERATION WITH INT OF UNSPECIFIED SIGNIFICANCE",0);'GOTO' S[6] 'END' DEFAULT FLOAT;
   'GOTO' S[A+A+B+2]
'END' SCALETEST;

'PROCEDURE' UNARYMINUS;'IF' TYP[RH]<>FLOATING 'THEN'
       'IF' TYPEBITS[RH]<0 'THEN' DIRADD[RH]←-DIRADD[RH] 'ELSE' GOODPICK(RH,1)
   'ELSE' 'IF' TYPEBITS[RH]<0 'THEN' DIRADD[RH]←DIRADD[RH]!DIFFER!#777777B0+#100
       'ELSE' 'BEGIN' GOODPICK(RH,0);OUTXFMN(RHACC,#16,0,OUTWCONST(-#100),"(UNARY)");
           OUTXFMN(RHACC,#06,0,#100,"(MINUS)")
'END' UNARYMINUS;
```

```
'PAGE' ARITHMETIC;

'PROCEDURE' ACCPICK('VAL' 'INT' REF,ACC);
'BEGIN' 'IF' ISINACC(REF)<>ACC 'AND' ACCS[ACC]<0 'THEN' DUMPACC(ACC);
    PICK(REF,ACC,STB[REF],0);ACCS[ACC]←0
'END' ACCPICK;

'PROCEDURE' FLOATIT('VAL' 'INT' REF);'IF' TYP[REF]<>FLOATING 'THEN' 'IF' TYPEBITS[REF]<0 'THEN' SCALENUM(REF,#10000)
'ELSE' 'BEGIN' ACCPICK(REF,7);DUMPACCS;ZEROACCS;OUTI(#1420,STB[REF],"(TYPE)");
    CALLLIB(10,"(FLOAT)");TYPEBITS[REF]←#10000;ACCUPDATE(REF)
'END' FLOATIT;

'PROCEDURE' FLOATOP('VAL' 'INT' OP);'BEGIN' 'IF' OP<>6 'THEN' FLOATIT(RN);FLOATIT(LN);
    'IF' RNACC=7 'THEN' 'BEGIN' PERN;'IF' OP>=3 'THEN' OP←OP+4 'END' ;ACCPICK(LN,7);ACCPICK(RN,6);
    DUMPACCS;ZEROACCS;CALLLIB(10+OP,"(FP OP)");ACCUPDATE(LN);'GOTO' BENEADEXIT
'END' FLOATOP;

'PROCEDURE' ADDSUB('VAL' 'INT' FUN);'IF' EXPSCALE=#10000 'THEN' FLOATOP(FUN) 'ELSE'
'BEGIN' PICK(LN,GOODACC(LN),EXPSCALE,0);
   'IF' TYPEBITS[RN]<0 'THEN' SCALENUM(RN,EXPSCALE) 'ELSE'
   'IF' PARTWORD[RN]<>0 'OR' 'BITS'[6,18](TYPEBITS[RN] 'DIFFER' EXPSCALE)<>0 'THEN' PICK(RN,GOODACC(RN),EXPSCALE,0);
   'INST'(RN,LNACC,FUN);TYPEBITS[LN]←EXPSCALE;'GOTO' BENEADEXIT
'END' ADDSUB;

'INT' 'PROC' POWERTWO;'ANSWER'
   'IF' TYPEBITS[RN]<0 'AND' TYP[RN]=0 'AND' CYCLE(DIRADD[RN],26-SGB[RN])=1'THEN' SGB[RN]←2 'ELSE' 0;

'PROCEDURE' FIXCON;'IF' TYPEBITS[RN]<0 'AND' TYP[RN]=0 'THEN' SCALENUM(RN,#7037+SGB[RN]);

'INT' 'PROC' SETASS;'BEGIN' ASSFUN←#10;EXPSCALE←STB[RN];SCALEFIRM←1;
    'IF' 'BIT'[3]TRACE<>0 'THEN' 'BEGIN' PREFACC←7;'ANSWER' 1 'END'
    'ELSE' 'BEGIN' PREFACC←FINDACC(7);'ANSWER' 0 'END'
'END' SETASS;

'INT' 'PROC' LNEQRN;'ANSWER' 'IF' TYPEBITS[LN]<>TYPEBITS[RN] 'OR' TYP[LN]=FLOATING 'OR' PARTWORD[LN]<>0
    'OR' PARTWORD[RN]<>0 'OR' DIRADD[LN]<>DIRADD[RN] 'OR' INDADD[LN]<>INDADD[RN] 'THEN' 0 'ELSE' 1;

'INT' 'PROC' SELOPTA;'BEGIN'
    'IF' ASSFUN<0 'THEN' 'BEGIN' ASSFUN←ASSFUN-MARK;'IF' ASSFUN=#10 'THEN' UNARYMINUS 'END' ;
    'IF' ASSFUN=#10 'THEN' 'ANSWER' 0 'ELSE' 'BEGIN' BENEAD;'ANSWER' 1 'END'
'END' SELOPTA;
```



```
'PAGE' ASSIGNMENT AND FOR;

'PROCEDURE' STOREAWAY('VAL' 'INT' TFLAG);'IF' ASSFUN=0 'THEN' BENEAD 'ELSE' 'BEGIN' 'INT' ADD,ACC,FFLAG;
   FFLAG←'IF' TFLAG=0 'AND' PARTWORD[LN]<>0 'AND' TYPEBITS[RN]<0 'THEN' 1 'ELSE' 0;
   ADD←'IF' FFLAG=0 'AND' PARTWORD[LN]<>0 'OR' INDADD[LN]<>0 'OR' ASSFUN<>#10 'THEN'
       ('IF' TYPEBITS[LN]=TYPEBITS[RN] 'AND' INDADD[RN]=0 'AND' PARTWORD[RN]=0 'AND' DIRADD[RN]>=0 'THEN' DIRADD[RN]
           'ELSE' 0) 'ELSE' DIRADD[LN];
XX;ACC←'IF' TFLAG<>0 'THEN' 7 'ELSE' 'IF' ASSFUN<0 'THEN' 0 'ELSE' 'IF' COPYINACC(RN)<>0 'THEN' COPYINACC(RN)
       'ELSE' 'IF' COPYINACC(LN)<>0 'THEN' COPYINACC(LN) 'ELSE' FINDACC(7);
    'IF' FFLAG<>0 'THEN' 'BEGIN' 'IF' INDADD[LN]<>0 'THEN' ACCS[ACC]←-1 'END'
       'ELSE' 'IF' ACC<>0 'THEN' PICK(RN,ACC,STB[LN],0);
    'IF' TFLAG<>0 'THEN' 'BEGIN' 'IF' INDADD[LN]<0 'THEN' DUMPACC(INDADD[LN]-MARK);ZEROACCS;
       OUTI(#1420,EXPSCALE,"(TYPE)");OUTI(#1220,STA,SPIEL[LN]);DOSTRING(SPIEL[LN],OUT24CS);CALLLIB(7,"(TRACE)") 'END';
    'IF' PARTWORD[LN]=0 'THEN' INST(LN,ACC,ASSFUN 'MASK' #37)
    'ELSE' 'BEGIN' 'INT' SHIFT,MOD,MOD←ISMOD(LN);
       SHIFT←'IF' TYP[LN]=INTEGER 'THEN' LSB[LN] 'ELSE' ('IF' PARTWORD[LN]<0 'THEN' 1 'ELSE' 0)-MSB[LN];
       'IF' FFLAG=0 'THEN' 'BEGIN' 'IF' SHIFT<0 'THEN'
              OUTXFMN(ACC,'IF' SHIFT>=0 'THEN' #33 'ELSE' #32,0,'IF' SHIFT>=0 'THEN' SHIFT 'ELSE' -SHIFT,"(ALIGN)");
           MASKINST(LN,ACC,0);OUTXFMN(ACC,#14,MOD,DIRADD[LN],SPIEL[LN]);
           MASKINST(LN,ACC,-1);OUTXFMN(ACC,#12,MOD,DIRADD[LN],"(PACK)")
       'END' PACKING 'ELSE' 'BEGIN' 'INT' F;F←CYCLE(RESCALE(DIRADD[RN],TYPEBITS[RN],EXPSCALE),
                 'IF' SHIFT>=0 'THEN' SHIFT 'ELSE' 24-SHIFT)'MASK' FORMMASK(RN);
           'IF' COPYINACC(LN)<>ACC 'THEN' OUTXFMN(ACC,00,MOD,DIRADD[LN],SPIEL[LN]);
           'IF' F<>FORMMASK(LN) 'THEN' MASKINST(LN,ACC,-1);
           'IF' F<>0 'THEN' OUTXFMN(ACC,#17,0,OUTCONST(F),"(CONST)");
           OUTXFMN(ACC,#10,MOD,DIRADD[LN],SPIEL[LN])
           'END' FIELD BECOMES CONSTANT;
       'END' PARTWORD ASSIGNMENT;
    'BEGIN' 'INT' A;'FOR' A←COPYINACC(LN) 'WHILE' A<>0 'DO' ACCS[A]←0 'END' KILL OLD COPIES;
    BENEAD;( KILL RN ) ACCS[ACC]←ADD; ( UPDATE )
'END' STOREAWAY;

'PROCEDURE' SETRT;'BEGIN' 'IF' RTA<>0 'THEN' OUTS(35,RTA);RTA←STA;OUT24CS(0) 'END' SETRT;

'PROCEDURE' UJBACK;OUTI(#100,BJA,"(REPT)");

'PROCEDURE' SUSPIEL;'BEGIN' SPIEL[LN]←"(STEP)"ISPIEL[RN]←"(LIMIT)" 'END' SUSPIEL;

'PROCEDURE' FORTEST;'BEGIN' SUSPIEL;INST(CV,P,'IF' TYPEBITS[LN] 'MASK' DIRADD[LN]>=0 'THEN' 01 'ELSE' 00);
  'IF' TYPEBITS[RN]>=0 'OR' DIRADD[RN]<>0 'THEN' INST(RN,7,'IF' TYPEBITS[LN] 'MASK' DIRADD[LN]>=0 'THEN' 02 'ELSE' 03);
  'IF' TYPEBITS[LN]>=0 'THEN' INST(LN,7,#36);
  SKIPCHAIN←PTA;OUTI(#1714,0,"(TEST)");BENEAD
'END' FORTEST;

'PROCEDURE' FORINC;'BEGIN' ASSFUN←#12;STOREAWAY(0) 'END' FORINC;

'PROCEDURE' FORCON;'BEGIN' FORSTATE←FORSTATE 'MASK' 3;'IF' FORSTATE=2 'THEN' FORTEST;
    OUT27(RTA,"(0)");ZEROACCS;'IF' FORSTATE=2 'THEN' FORINC;'IF' FORSTATE<>0 'THEN' UJBACK;FORSTATE←4
'END' FORCON;
```

```
'PAGE' PROCEDURE CALLS;

'INT' 'PROC' RNSPTEST;'ANSWER' 'IF' TYP[RN]>=INTPROC 'THEN' 1 'ELSE' 0;

'PROCEDURE' MAKEPARAM('VAL' 'INT' TYPE);'BEGIN'
      GRABVAR;TYPEBITS[RN]←TYPEBITS[PSP] 'MASK' #07000000 + #67;PSP←PSP+1;SPIEL[RN]←'('(TYPE)');
      'IF' TYPE=0 'THEN' TOPACC(7)
      'ELSE' 'BEGIN' TYPEBITS[RN]←TYPEBITS[RN]←MARK;DIRADD[RN]←'IF' TYPE=2 'THEN' EXPSCALE 'ELSE' TYPEBITS[PNP] 'END'
'END' MAKEPARAM;

'INT' 'PROC' SETUPPROC;'BEGIN' DUMPACCS;'IF' PARAMSPEC[RN]<0 'THEN' STOPOP('LOC'(NAME),"RECURSIVE CALL");
   ONSTACK(8,EXPRCHAIN);PNP←RN;FPP←STACKPOINTER;SETLNRN;PSP←PARAMSPEC[PNP]-1;
   'IF' SPB[PNP]=0 'THEN' TYPEBITS[PNP]←STB[PNP]-#24000 'ELSE'
      'BEGIN' TYPEBITS[PNP]←'IF' SCALEFIRM<>0 'THEN' EXPSCALE 'ELSE' #10000;PNP←PNP+MARK;MAKEPARAM(1) 'END' ;
   'ANSWER' 'IF' TYPEBITS[PSP]<0 'THEN' 0 'ELSE' 1
'END' SETUPPROC;

'INT' 'PROC' PARAMCLASS;'ANSWER' 'IF' LCM[PSP]<>0 'THEN'('IF' AYM[PSP]<>0 'THEN' 2 'ELSE' 1)
      'ELSE' 'IF' TYP[PSP]=LABEL 'THEN' 3 'ELSE' 'IF' TYP[PSP]>=PROCEDURE 'THEN' 2 'ELSE' 0;

'INT' 'PROC' NEXTPTYPE;'BEGIN' PREACC←PAC[PSP];'IF' SPB[PSP]<>1 'THEN'
   'BEGIN' SCALEFIRM←1;EXPSCALE←STB[PSP] 'END' 'ELSE' 'BEGIN' SCALEFIRM←0;EXPSCALE←0 'END' ;
'ANSWER' PARAMCLASS 'END' NEXTPTYPE;

'INT' 'PROC' ANYMORE;'BEGIN' 'SWITCH' CLASS←C0,C1,C2,C3;
   'IF' SPB[PSP]=1 'THEN' 'BEGIN' EXPSCALE←STB[RN]]'IF' T=5 'THEN' EXPSCALE←EXPSCALE+AMARK 'END' ;
   'GOTO' CLASS[PARAMCLASS+1];
ER;STOPOP(SPIEL[RN],"IS WRONG PARAMETER TYPE");'GOTO' C3;
C0;'IF' (TYPEBITS[RN] 'DIFFER' EXPSCALE) 'MASK' #34077 <> 0 'THEN' 'IF' TYPEBITS[RN]<0 'THEN' SCALENUM(RN,EXPSCALE)
      'ELSE' PICK(RN,GOODACC(RN),EXPSCALE,0); 'GOTO' C3;
C2;'IF' TYP[RN]>=PROCEDURE 'THEN' 'BEGIN' 'IF' PARAMSPEC[RN]<0 'THEN' STOPOP(SPIEL[RN],"RECURSIVE USE");PARAMSPEC[RN]←U;
   'IF' SPB[RN]<>SPB[PSP] 'THEN' 'GOTO' ER 'ELSE' 'IF' TYP[PSP]=PROCEDURE 'THEN' 'GOTO' C3 'END' ;
C1;'IF' STB[RN]<>EXPSCALE 'OR' AYM[PSP]<>AYM[RN] 'OR' PARTWORD[RN]<>0 'THEN' 'GOTO' ER;
C3;'IF' LCM[PSP]<>0 'THEN' LOCACT;
   'IF' INDADD[RN]<0 'THEN' GOODPICK(RN,0); PAC[RN]←PREPACC;
      PSP←PSP+1;'IF' SPB[PSP]=2 'THEN' MAKEPARAM(2);'ANSWER' 'IF' TYPEBITS[PSP]<0 'THEN' 0 'ELSE' 1
'END' ANYMORE;

'INT' 'PROC' INSTTYPE;'BEGIN' 'SPECIAL' 'ARRAY' S←#00004000,#11111000,#33333311,#22222100;
    'IF' FUNCTION=5 'THEN' 'BEGIN' FUNCTION←4;'ANSWER' 3 'END' ;
    'ANSWER' 'BITS'(3,21)CYCLE(S['BITS'(2,19)FUNCTION],'BITS'(3,21)FUNCTION+3+3)
'END' INSTTYPE;
```



```
'PAGE' SYNTAX 1;

'INTEGER' SSEL,SSPTR;'INTEGER' 'ARRAY' SS[0:99];

'SPECIAL' 'ARRAY' SYNTAX←#600:READER,#400/5,#50602027,#50130001,#200:FAIL,#405000074,#40560530,#41061526,#50621530,#70700
016,#70300000,#400/907,#50312027,#200:FAIL;#70710023,#70300000,#400/955,#50312027,#200:FAIL;#71070067,#600:SETTEST,#4050
0074,#40560530,#70620000,#600:CLEARTYPE,#61130047,#400/107,#60560041,#400/902,#70770000,#400/174,#700/SCALECON,#200:SETL
18VAR,#600:TYPETPROC,#70550000,#400/902,#600:SETLIBSEG,#400/387,#300/FINISHSEG,#60660060,#400/227,#400/902,#600:SETLIBSE
G,#70120000,#400/148,#600:NEXTPSET,#400/387,#300/FINISHSEG;#600:SETYES,#600:TYPEPROC,#70550000,#400/902,#600:SETLIBSEG,#
400/387,#300/FINISHSEG,#61102027,#400/580,#400/857,#50172027,#200:FAIL,#70300000,#70300000,#600:COMON,#400/67,#600:COMOF
F,#50312027,#200:FAIL,#600:STARTDEC,#400/71,#50130103,#200:EXIT,#40720425,#40270453,#70560116,#600:TYPELAB,#400/224,#503
40113,#200:EXIT,#70610121,#600:TYPEISWITCH,#000/75;#61130140,#400/107,#61210132,#400/132,#70772027,#600:SKIPDTA,#400/166
,#50340127,#200:EXIT,#600:TYPETPROC,#70550000,#400/224,#400/182,#50340134,#200:EXIT,#60660143,#400/227,#000/92,#60640146
,#400/231,#000/85,#60552027,#600:SETYES,#600:TYPEPROC,#50550134,#200:FAIL,#50510167,#70520165,#71270000,#71160000,#600:N
OBITS,#70340000,#400/125,#600:NOAFTER,#52422027,#200:FAIL,#70530000,#200:TYPEFLOAT,#70730174,#71160000,#600:NOBITS,#7073
0000,#200:PARTINT,#200:TYPEINT,#51162027,#70330201,#51162027,#200:FAIL,#70350000,#71160000,#200:NEGNUM,#70410217,#600:TY
PEARRAY,#600:ZERORARRAYS,#400/145,#70730000,#400/150,#400/155,#70750000,#600:ENDARRAY,#50340206,#200:EXIT,#400/145,#200:D
EFSIZE,#400/148,#50340221,#200:EXIT,#70560000,#200:NEWNAME,#400/125,#600:SETLB,#70120000,#400/125,#200:SETUB,#60340245,#
600,FIRSTDIM,#70340000,#400/150,#60340244,#600:MIDDIM,#70340000,#400/150,#400/159,#200:LASTDIM,#200:ONEDIM,#70050250,#20
0:PRESETSTRING,#70300254,#400/87,#50312027,#200:FAIL,#400/174,#200:OUTPRESET,#51242027,#70330262,#51242027,#200:FAIL,#70
350265,#71240000,#200:NEGNUM,#200:ZERONUM,#600:BEGINPSPEC,#70300273,#400/188,#70310000,#200:ENDPSPEC,#200:ENDPSPEC,#400/
193,#700/NEXTPARAM,#70342027,#600:NEXTPSET,#000/188,#50660331,#70250307,#600:TYPELOC,#61130153,#600:TYPESPEC,#300/NEXTPA
RAM,#70540311,#200:TYPELAB,#70610313,#200:TYPEISWITCH,#70640316,#600:TYPEINT,#200:TYPEIARRAY,#61130325,#400/107,#7041032
2,#200:TYPEIARRAY,#600:TYPETPROC,#50552027,#200:FAIL,#600:SETYES,#600:TYPEPROC,#50552027,#200:FAIL,#61130153,#600:50330336,#
600:TYPEVPROC,#50552027,#200:FAIL,#600:TYPESPEC,#300/NEXTPARAM,#600:ADDRSPEC,#400/148,#200:SPECONE,#70660000,#600:TYPEVP
ROC,#50552027,#200:FAIL,#70640000,#600:TYPEINT,#600:TYPEARRAY,#400/148,#707730000,#7116000,#600:TABLESIZE,#707500000,#400
/242,#600:CLEARTYPE,#200:TYPEINT,#70730000,#600:TARRAY,#600:CLEARTYPE,#70500000,#400/252,#600:TYPEARRAY,#600:NEWNAME,#50
752027,#50130422,#200:FAIL,#41270403,#70570400,#400/259,#200:UNSFIELD,#400/107,#400/125,#200:FIELDDISP,#71270000,#711600
00,#600:NOBITS,#600:NOSIG,#400/270,#72420000,#400/125,#600:FIELDDISP,#72340000,#71160000,#200:FIELDPOSN,#70340421,#400/1
25,#200:NOAFTER,#200:PARTINT,#41320371,#60560364,#200:FAIL,#70720000,#400/283,#71000000,#600:OVERON,#400/296,#200:OVEROF
F,#70560442,#600:LOOKUP,#70732027,#400/125,#600:INCTOS,#50752027,#200:FAIL,#700/NONAME,#70730000,#71160000,#600:INCTOS,
#50752027,#200:FAIL,#40640347,#400/107,#000/132,#70270000,#70410000,#600:TYPEINT,#600:TYPEARRAY,#600:ADDRSPEC,#400/148,#
70770000,#400/309,#50340462,#200:EXIT,#70050467,#300/SPECSTRING,#61130502,#600:CLEARTYPE,#400/107,#70300000,#400/174,#60
0,SPECNUM,#50312027,#70340000;#400/174,#600:SPECNUM,#000/317,#400/335,#61400514,#600:SETTEN,#50370521,#70120511,#7050000
0,#200:SPECLAB,#71430000,#400/125,#200:SPECREL,#600:CLEARTYPE,#600:TYPEINT,#200:SPECNUM,#413,0175,#200:ZERONUM,#613,0546
,#400/125,#700/NONAME,#600:INCTOS,#200:SPECCON,#400/283,#200:SPECCON,#70360000,#600:BEGINPROC,#70420000,#400/351,#600:EN
DPROC,#50102027,#200:FAIL,#400/356,#600:ENDDECS,#400/524,#50130541,#200:EXIT,#600:STARTDEC,#400/361,#70130000,#41540344,
#200:EXIT,#40720425,#40270453,#41061526,#70160561,#600:SETNO,#400/857,#50172027,#200:FAIL,#70610972,#600:ADDRSW,#600:TYP
ESWITCH,#400/148,#70770000,#70560000,#600:SPECLAB,#50340566,#200:EXIT,#61130607,#400/107,#61210577,#400/132,#000/85,#600
:TYPETPROC,#70550000,#400/148,#600:BEGINPROC,#400/417,#70130000,#400/477,#200:ENDPROC,#60660617,#400/227,#400/148,#600:B
EGINPROC,#70120000,#400/148,#600:NEXTPSET,#000/387,#60640622,#400/231,#000/85,#600:SETYES,#600:TYPEPROC,#50551341,#70200
631,#600:ASSTRACE,#50172027,#200:FAIL,#70210635,#600:FORTRACE,#50172027,#200:FAIL,#70540000,#600:LABTRACE,#50172027,#200
,FAIL,#70302027,#400/421,#50312027,#200:FAIL,#400/425,#70132027,#600,NEXTPSET,#000/421,#30660724,#70250665,#600:TYPELOC,
#41130657,#400/107,#000/145,#600:TYPESPEC,#400/148,#70120000,#400/148,#50340660,#200:EXIT,#70540670,#600:TYPELAB,#000/14
5,#70610673,#600:TYPEISWITCH,#000/145,#70640705,#600:TYPEINT,#600:TYPEIARRAY,#400/148,#70730000,#71160000,#70750000,#600
:PARAMTAB,#400/242,#200:PARAMTAB,#61130720,#400/107,#70410712,#600:TYPEIARRAY,#000/145,#600:TYPETPROC,#70550000,#400/148
,#400/182,#50340714,#200:EXIT,#600:SETYES,#600:TYPEPROC,#50350714,#200:FAIL,#61130727,#400/107,#000/145,#60560732,#600:T
YPESPEC,#000/432,#600:TYPEVPROC,#50550714,#200:FAIL,#70430745,#70420000,#600:KILLPARAMS,#400/496,#700/ENTERPROC,#400/500
,#50102027,#200:FAIL,#600:SETPARAMS,#70420754,#400/496,#600:PROCENTRY,#400/353,#70100000,#200:EXITCHECK,#600:ENDDECS,#60
0,PROCENTRY,#400/842,#200:EXITCHECK,#61340763,#400/356,#200:ENDDECS,#200:ENDDECS,#400/503,#50130764,#700:EXIT,#70060773,
#700/INSTTYPE,#400/1023,#200:OUTCODE,#70560777,#600:CODELAB,#50120767,#200:FAIL,#60422027,#400/514,#300/STATUSCODE,#700/
STATUSCHECK,#70420000,#61571010,#400/353,#50102027;#200:FAIL,#700/BEGINBLOCK,#400/351,#70100000,#300/ENDBLOCK,#400/526,#
200:ENDST,#40111052,#40462103,#40741443,#40471450,#40421002,#30361401,#70431041,#700/STATUSCODE,#70420000,#02121033,#400
/500,#50102027,#200:FAIL,#700/BEGINBLOCK,#400/356,#600:ENDDECS,#400/500,#70100000,#300/ENDBLOCK,#61622027,#700/STATUSCHE
CK,#400/831,#70770000,#600:VARCHECK,#700/SETASS,#400/1069,#600:STORE,#300/BENEAD,#70110000,#600:IFS,#400/564,#70140000,#
600,ENDST,#400/524,#70431063,#600:ELSES,#400/526,#200:FI,#400/371,#70221070,#700/OUTCJ,#000/584,#70232027,#600,ORACT,#00
0/564,#61651077,#400/580,#701,0000,#200:OVRTEST,#700/STACKEXPR,#600:ANYPREF,#400/584,#400/782,#200:RELATION,#701411000,#2
00:SETNO,#600:SETYES,#200:TYPEPROC,#41731113,#70000000,#200:CONSTANT,#42011117,#400/598,#42011400,#200:EXIT,#70531122,#4
00/598,#000/589,#70350000,#400/598,#700/UNARYMINUS,#000/589,#400/661,#400/766,#200:SCALETERM,#70561135,#700/LOOKUP,#4073
1141,#200:ANYCHECK,#700/NONAME,#400/613,#70750000,#300/KILLEXPR,#400/613,#400/652,#70750000,#300/KILLEXPR,#70730000,#600
:SETUPSUB,#42011160,#50002027,#60111154,#400/632,#200:PLUSSUB,#400/598,#600:PLUSSUB,#42011160,#200:EXIT,
```

```
#70331164,#400/598,#600:PLUSSUB,#000/622,#70330000:#400/598,#600:MINUSSUB,#000/622,#70110000,#600:IFEX,#400/564,#70140000,#400:THENEX,#400/644,#70450000,#600:ELSEFI,#600:ELSEX,#400/644,#600:ELSEFI,#200:FIEX,#40111170,#41731113,#70001211,#600:CONSTANT,#200:SCALETERM,#70050000,#600:STRINGEX,#200:SCALETERM,#70362027,#600:SUBCOMMA,#400/615,#000/652,#60731224,#700/LOOKUP,#400/606,#200:LABSX,#200:LABL,#400/660,#70762027,#400/666,#600:RAISE,#000/662,#400/671,#70072027,#400/671,#600:NFO,#000/667,#400/676,#70242027,#400/676,#600:ORF,#000/672,#400/681,#70262027,#400/681,#600:NBX,#000/677,#61021260,#400/690,#70052027,#600:SETSHIFT,#400/756,#400:DOSHIFT,#000/683,#400/711,#000/683,#400/694,#400:BITBIN,#400/711,#200:RMSBITS,#71020000,#600:CLEARTYPE,#70730000,#71160000,#400/703,#600:FIELDPOSN,#600:UNSFIELD,#50752027,#200:FAIL,#60361306,#600:MOBITS,#600:NOSIG,#600:PARITNT,#70340000,#51162027,#200:FAIL,#200:ONEBIT,#62071320,#70021312,#200:CONSTANT,#70300000,#700/STACKEXPR,#600:SETPREF,#400/644,#70310000,#200:OFFSTEXPR,#70011322,#200:CONSTANT,#70361336,#700/LOOKUP,#60731327,#400/600,#200:VARCHECK,#700/RMSPTEST,#100/2,#200:VARCHECK,#700/SETUPPROC,#400/1011,#600:CALLPROC,#200:FINISHPROC,#70251344,#700000,#400/601,#700/LOCACT,#50312027,#200:FAIL,#70541352,#70300000,#70360000,#600/656,#50312027,#200:FAIL,#611131362,#600:CLPARTYPE,#400/107,#600:TYPEXPR,#70300000,#600/644,#70310000,#200:EXPRTYPE,#700/NONAME,#000/606,#70301370,#400/615,#50312027,#200:FAIL,#61021373,#400/690,#200:PLUSSUB,#400/720,#600:SCALETERM,#200:PLUSSUB,#70321402,#400/661,#600:MPY,#000/766,#70372027,#400/661,#600:DIVIDE,#000/766,#70331612,#400/598,#600:ADD,#000/589,#70350000,#400/598,#600:SUB,#000/589,#5041420,#200:SETNEZ,#70001422,#200:ZEROCOMP,#400/795,#70331627,#400/598,#600:CONDADD,#000/787,#70352027,#400/598,#600:CONDSUB,#000/787,#70331436,#400/598,#200:CONDPLUS,#70351441,#400/598,#200:CONDMINUS,#400/598,#200:CONDPLUS,#700/STATUSCHECK,#70740000,#600:SETANS,#400/644,#200:ANSCHECK,#704/0000,#70560000,#60731460,#700/LOOKUP,#700/STATUSCHECK,#400/606,#600:LABEL,#200:GOTOSK,#200:GOTOL,#60121465,#600:SETLAB,#50121016,#200:FAIL,#700/LOOKUP,#700/STATUSCHECK,#62151472,#400/829,#000/548,#600:LHSPROC,#400/730,#300/REHEAD,#40731141,#200:EXIT,#61021510,#400/694,#70561305,#700/LOOKUP,#600:LHSBITS,#000/829,#700/NONAME,#600:LHSBITS,#000/606,#700/NONAME,#000/606,#40111052,#40462105,#40741443,#40471430,#70561322,#700/LOOKUP,#700/STATUSCHECK,#000/823,#61622027,#700/STATUSCHECK,#400/831,#000/548,#71060000,#71160000,#200:SETLEVEL,#70201533,#200:ASSTRACE,#70211535,#200:FORTRACE,#70541537,#200:LABTRACE,#70550000,#200:PROCTRACE,#60561565,#400/148,#600:BEGINPROC,#000/387,#600:PROCTRACE,#50172027,#200:FAIL,#70301554,#400/878,#50312027,#200:FAIL,#600:CLEARTYPE,#000/25,#600:CLEARTYPE,#400/882,#50131556,#200:EXIT,#611131573,#400/107,#40561603,#600:TYPETPROC,#70550000,#400/902,#400/182,#50361567,#200:EXIT,#60661576,#400/227,#000/887,#60552027,#600:SETYES,#600:TYPEPROC,#50551567,#200:FAIL,#400/902,#50341603,#200:EXIT,#70560000,#70370000,#71160000,#600:LIBADD,#200:NEWNAME,#600:CLEARTYPE,#400/911,#50131613,#200:EXIT,#70561624,#600:TYPELAB,#400/948,#50341621,#200:EXIT,#70611627,#600:TYPESWITCH,#000/913,#70641637,#600:TYPEINT,#600:TYPEARRAY,#400/948,#70730000,#71160000,#50730362,#200:FAIL,#611131652,#400/107,#61211644,#400/946,#000/913,#600:TYPETPROC,#70550000,#400/948,#400/182,#53341646,#200:EXIT,#60661655,#400/227,#000/934,#60552027,#600:SETYES,#600:TYPEPROC,#50551646,#200:FAIL,#70412027,#200:TYPEARRAY,#70560000,#70370000,#600:SETTEN,#71160000,#400/335,#600:EXTAPP,#200:NEWNAME,#600:CLEARTYPE,#400/959,#50131673,#200:EXIT,#70541704,#600:TYPELAB,#400/994,#50341701,#200:EXIT,#70611707,#600:TYPESWITCH,#000/961,#70641717,#600:TYPEINT,#600:TYPEARRAY,#400/994,#70730000,#71160000,#50730362,#200:FAIL,#611131732,#400/107,#61211724,#400/946,#000/961,#600:TYPETPROC,#70550000,#400/994,#400/182,#50341726,#200:EXIT,#60661735,#400/227,#000/982,#60552027,#600:SETYES,#600:TYPEPROC,#50551726,#200:FAIL,#70560000,#70370000,#71160000,#600:ABSADD,#200:NEWNAME,#700/NEXTPTYPE,#400/1015,#700/ANYMORE,#100/2,#000/1006,#50341747,#200:FAIL,#50312027,#70340000,#600:CALLPROC,#600:MULTICALL,#000/999,#100/2,#200:EXIT,#50301747,#201:FAIL,#100/4,#000/644,#000/601,#000/1021,#50561220,#200:FAIL?#70560000,#300/LOOKUP,#100/5,#000/1058,#000/1062,#000/1036,#000/1031,#42232022,#70050000,#200:STRINGEX,#70362011,#200,LABL,#71430000,#400/335,#200:RELADD,#71162016,#200:CODESHIFT,#70600000,#400/335,#600:CODESHIFT,#200:NISMOD,#70402024,#200:NISACC,#70562030,#600:LOOKUPD,#40732031,#200:EXIT,#700/NONAME,#707300000,#400/1054,#600:INCTOS,#50752027,#200:FAIL,#41350175,#70600000,#600:NISMOD,#000/335,#42232022,#61352046,#400/125,#200:CONSTANT,#600:CLEARTYPE,#400/107,#70300000,#400/174,#700/SCALECON,#70310000,#200:CONSTANT,#100/2,#000/1072,#000/644,#40111170,#70052063,#200:STRINGEX?#70002065,#200:STOREZERO,#70332076,#400/598,#600:ADDA,#62012075,#700/SELOPYA,#100/2,#000/774,#000/591,#200:SIMPLEASS,#70352102,#400/598,#600:SUBA,#000/1080,#400/598,#600:ADDA,#000/1080,#70460000,#600:STARTFOR,#400/601,#600:CHECKCV,#70770000,#400/1072,#600:ASSCV,#42262120,#70672127,#400/564,#600:WHILEL,#70442124,#600:DOCS,#400/526,#200:ENDFOR,#70340000,#600:MOREFOR,#000/1098,#72310000,#400/644,#600:STEPUNT,#72370000,#400/644,#600:STEPUNT,#000/1104
;

   'SPECIAL' 'ARRAY' BOOLWORD←#00001300,#00000014,#00000000,#00000000,#00000160,#00000000,#60000000,#00000000,#00000000,
#00000000,#00004002,#00000000;#70000000,#00000000,#00000000,#00000000,#40000000,#00010000,#00010000,#00000000,#00002000,
#60000000,#05000000,#00000000,#00020000,#10600000,#00000000,#00000000,#10400000,#00000000,#00000000,#00000002,#00010000,
#00000300,#00000014,#00000000;#00001301,#00000174,#22420002,#00150000,#00031402,#00014040,#00000000,#00000000,#00010040,
#00003000,#00000000,#00000000,#70000004,#43000172,#00010040,#30000004,#43000172,#00010040,#30000004,#40000172,#00010040,
#00000000,#05000000,#00000000;#30000004,#40000172,#00010000,#70000006,#00000172,#00010000,#00510000,#0002000Z,#00000000,
#00000000,#00000000,#00010400,#00000000,#00000164,#00460000,#00000000,#00100002,#00010000,#00000000,#02004000,#00000000,
#00020000,#00000000,#04000000,#00000000,#02000000,#00000040,#00020000,#00000000,#01000000,#00000000,#20000000,#00002000;
(****)
```



```
STARTUP:CSUM←0;LINK←0;DECLIST←0;LOCALLIMIT←0;BITSPEC←0;INDADDE0]←0;MCTOP←MCSTART;

STACKPOINTER←STACKSTART;MCCHAIN←0;MCSOURCE←0;MCBODY←0;MCLIST←0;MCFLAG←0;

LABSTACKPTR←STACKFINISH;PRESETOK←1;COMMON←0;OVERLAY←0;TEST←0;

TRACE←0;LABDECLIST←0;LEVEL←1;'FOR' INCTR←63;←1;1 'DO' INBUFF[INCTR]←←1;

HALT("GO TO COMPILE");T2←←1;
```

```
'PAGE' ANALYSER;


'CODE' 'BEGIN'
        1 LDX 0;                ( SYNTAX POINTER=0 )
        0 STO SSPTR;            ( SYNTAX STACK POINTER=0 )
RPPT:   7 LDX SYNTAX[B1];       ( COPY WORD OF SYNTAX )
        1 ADD 1;                ( STEP POINTER )
CHEAT:  2 LDX B7;               ( COPY WORD )
        7 SLC 1;                ( BIT 1 TO SIGN )
        2 JGE NOTEST;           ( JUMP IF NOT TEST OF T.S. )
        4 LDX B2;               ( COPY WORD )
        4 AND #7777;            ( MASK DESTINATION FIELD )
        2 SRL 12;
        2 AND #777;             ( MASK B FIELD )
        5 LDX B2;
        5 SUB T1;               ( SINGLE SYMBOL TEST )
        5 JZE YES;              ( JUMP IF T1=B )
        2 SUB T2;               ( SUBTRACT BOOLEAN BIAS )
        2 JLT NO;               ( JUMP IF SINGLE SYMBOL TEST )
        3 LDX T1;               (.INCOMING SYMBOL )
        3 SRL 23;
        3 DIV 24;               ( X=SHIFTS, Q=DISPLACEMENT )
        2 ADD [2];
        5 LDX BOOLWORD[B2];( COPY BOOLEAN WORD )
        5 SLC B3;
        5 JGE NO;               ( IF BIT 0=0, TEST FAILS )
YES;    7 JLT *2;               ( SKIP IF BIT 1 SET (IFJ) )
        1 LDX B6;               ( COPY DESTINATION TO SYNTAX POINTER )
        7 SLC 1;                ( BIT 2 TO SIGN )
        7 JGE REPT;             ( CONTINUE IF NOT ACCEPT T1 )
        7 LDX SYNTAX[0];        ( COPY #600/READER )
        0 JZE CHEAT;            ( INTERPRET B7 AS CALL ON READER )
NO:     7 JGE REPT;             ( CONTINUE IF (1TJ) )
        4 JNZ OK;  FAIL;'BEGIN' GIVEUP("SYNTAX FAILURE-) 'END' ;
OK;     1 LDX B4;               ( COPY DESTINATION TO SYNTAX POINTER )
        0 JZE REPT;             ( CONTINUE )
NOTEST: 2 AND #37777;           ( MASK DESTINATION/ADDRESS )
        7 JGE GOTO;             ( IF NOT CALL )
        3 LDX SSPTR;            ( SYNTAX STACK POINTER )
        1 STO SS[B3];           ( STORE SYNTAX POINTER IN STACK )
        3 ADD 1;
        3 STO SSPTR;
        3 SUB 100;              ( TEST AGAINST LIMIT )
        3 JLT GOTO;  'BEGIN' GIVEUP("SYNTAX STACK OVERFLOW") 'END';
GOTO;   7 SLC 1;                ( BIT 2 TO SIGN )
        7 JLT ACT;              ( JUMP IF ACTION )
        7 SLC 1;                ( BIT 3 TO SIGN )
        1 ADD SSEL;             ( IN CASE OF SEMANTIC SELECTION )
        7 JLT REPT;             ( SEMANTICALLY SELECTED RULE )
        1 LDX B2;               ( COPY DESTINATION TO SYNTAX POINTER )
        0 JZE REPT;             ( CONTINUE )
ACT:    7 SLC 1;                ( BIT 3 TO SIGN )
        7 JGE *2;               ( JUMP IF LABEL )
        2 LDX [B2];             ( COPY PROCEDURE ADDRESS )
        4 JCS B2;               ( CALL/GOTO ACTION )
        7 STO SSEL;             ( STORE RESULT FOR SEMANTIC SELECTION )
EXIT:   2 LDX SSPTR;            ( SYNTAX STACK POINTER )
        2 SUB 1;
        1 LDX SS[B2];           ( RESTORE SYNTAX POINTER )
        2 STO SSPTR;
        2 JGE REPT;             ( CONTINUE IF SYNTAX NOT FINISHED )
'END' ANALYSER;TEXTLINE("COMPILATION COMPLETE");'GOTO' STARTUP;
```



```
'PAGE' READER 1;


READER;'BEGIN' 'INTEGER' FRAC;NODS,MAC;

  'SWITCH' ULWORD←RO,RO,RO,PJ,RO,RO,SH,SH,SH,CM,PG,HA,DF,DL,FI,OI,LI,HX;

  'SWITCH' CHARACTER←DD,DD,DD,DD,DD,DD,DD,DD,DD,DD,CN,SC,LS,ES,GS,GM,SP,FT,ST,OC,ID,ID,AM,PR,OK,OK,AS,OK,OK,OK,FT,OK,
                   AC,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,OK,CM,OK,OK,OK;

'INTEGER' 'PROCEDURE' READ;'BEGIN' 'INT' ANS;
 R;'IF' MCSOURCE=0 'THEN' 'BEGIN' ANS←READTAPE;'IF' ANS>=#100 'THEN' 'GOTO' R 'END' NO MACRO EXPANSION
   'ELSE' 'BEGIN' ANS←NEXTCHAR(MCSOURCE);'IF' ANS=0" 'AND' NEXTCHAR(MCSOURCE)<>0" 'THEN'
      'BEGIN' 'IF' MCFLAG<>0 'THEN' 'GOTO' RE;MOVE(6,STACK[MCCHAIN],MCCHAIN);ANS←T2;'IF' ANS<0 'THEN' 'GOTO' R
      'END' GO DOWN A LEVEL NOT PERMITTED IF DEFINING MACRO OR PARAMETER
   'END' MACRO EXPANSION;
   'IF' COMMON<>0 'THEN' CHECKSUM(CSUM,ANS);  'ANSWER' ANS
'END' READ;

'INTEGER' 'PROCEDURE' READT2;'BEGIN' REPT;T2←READ;'IF' T2=16 'THEN' 'GOTO' REPT;'ANS' T2 'END' READT2;

'INTEGER' 'PROCEDURE' READT1;'BEGIN' T1←READT2;'ANSWER' T1 'END' READT1;

'INTEGER' 'PROCEDURE' RECOG('VAL' 'INT' ULW);'BEGIN' 'INTEGER' B,SI,J,JJJ;( ULW=0 FOR FUNCTION, 1 FOR 'WORD' )

  'SPECIAL' 'ARRAY' ULWRDS←0=4,512=42,256=72,0=91,#125000345,#70753241,#54647524,#23600174,#55044147,#45752362,#37434544,
#65624576,#55346401,#74404445,#63647307,#64504556,#74140541,#42544576,#64624143,#45741740,#44025774,#44046555,#40750406,
#51484645,#62740765,#05405156?#45752556,#45044575,#26360174,#25020475,#14034142,#45547454,#10450175,#11064554,#75061751,
#66420241,#62717462,#64456241?#54753157,#03614475,#03035760,#74214341,#64315756,#74250355,#41635574,#20175101,#74510244,
#74115664,#65474562,#74513065?#01745756,#64315756,#74264464,#51547465,#63514756,#45447457,#33450174,#10020175,#13035644,
#74100454,#63657445,#70644562?#64615474,#70610174,#41056262,#41717441,#10626337,#56656445,#74711256,#02447622,#63674562,
#74746363,#91475655,#45364674,#20075602,#45751257,#74161062,#45635766,#43627505,#11660174,#66415465,#45746617,#42017442,
#64916463,#75026547,#51567442,#40017452,#03576274,#46105457,#41665156,#47745351,#04704344,#74525651,#63507527,#16670451,
#66407500,#50515645,#74671547,#02457515,#02047510,#37645774,#47344301,#75265703,#44457443,#59056051,#54457501,#55035756,
#74504556,#64752256,#57020274,#23054364,#41547536,#66026274,#15456204,#54417174,#72465457,#07741563,#01746104,#45544674,
#36044445,#00746300,#67516443,#50746107,#54024175,#17637521,#07404543,#51615474,#27620241,#75165475,#20000000;

 'SPECIAL' 'ARRAY' FUNCTIONS←0=2,768=16,#06445166,#74371156,#03656176,#16547074,#01165403,#41427405,#03447074,#00574376,
#66520362,#63742503,#43637427,#03476574,#22035464,#74230356,#72742172,#45742004,#45704376,#14045360,#71743615,#41035644,
#74154402,#44740263,#74121657,#03624674,#17036564,#74266662,#74266303,#63427413,#03654274,#03076202,#41743034,#74320764,
#02567411,#57741054,#02417431,#02437433,#02547434,#66743500;

 'SPECIAL' 'ARRAY' CHOOSE←0/FUNCTIONS,0/ULWRDS,1,3;

 'INT' 'PROC' READS;'BEGIN' S←NEXTCHAR(SI);'ANSWER' S 'END' READS;

SI←CHOOSE[ULW,T2 'MASK' CHOOSE[ULW+2]];('IF' SI=0 'THEN' 'GOTO' RT;)
L;JJ←0;
M;J←0;
N;'IF' READS<33 'THEN' 'BEGIN' J←S;JJ←JJ;'GOTO' N 'END'
  'ELSE' 'IF' S>=60 'THEN' 'BEGIN' 'IF' ULW=0 'OR' T2=9 'THEN' 'GOTO' ANS 'END'
  'ELSE' 'BEGIN' 'IF' T2=5 'THEN' 'BEGIN' READT2;'GOTO' L 'END' 'END' ;
  'IF' J<>0 'THEN' 'BEGIN' SI←CYCLE(CYCLE(SI,2)+J,22);'GOTO' M 'END' ;
  'IF' S>=60 'OR' ULW=0 'OR' T2<>5' 'OR' JJ<>0 'THEN' 'GOTO' RT;
0;';F' READS<33 'THEN' 'GOTO' RT 'ELSE' ';F' S<60 'THEN' 'GOTO' 0;

ANS;'ANSWER' CYCLE(S-60,6)+READS 'END' RECOG;
```

```
'PAGE' READER 2;

'INT' 'PROC' NOASSY('VAL' 'INT' BASE,LIM);'BEGIN' 'INT' ANS;ANS←T1;NODS←0;'IF' ANS>=BASE 'THEN' 'GOTO' FT;
   'FOR' NODS←NODS+1 'WHILE' T2<BASE 'AND' NODS<LIM 'DO' 'BEGIN' ANS←ANS*BASE+T2;READT2 'END' FOR;
   'IF' T2<BASE 'THEN' 'GOTO' FT 'ELSE' 'ANSWER' ANS
'END' NUMBER ASSEMBLY;

'PROC' ADDCHAR;'CODE' 'BEGIN' 'SPECIAL' 'ARRAY' SHIFTS←24,18,12,6;1LDXMCID;1ADD#01000000;4OVRFT;1STOMCID;
      1SRL20;2LRXD;2SLL2;7LDXMCID[@1];2JNZ←2;7LDX#20202020;2LDXSHIFTS[@2];7SRL@2;6LDXT1;6SRL6;7SLL@2;
      7STOMCID[@1];0JCS84 'END' ADDCHAR;

'INT' 'PROC' OCTALNO;'BEGIN' READT2;'ANSWER' NOASSY(8,8) 'END' OCTALNO;

'INT' 'PROC' OCTALFRAC;'BEGIN' READT1;'ANSWER' CYCLE(OCTALNO,(8-NODS)←3) 'END' OCTAL FRACTION;

'PROCEDURE' GENOCT;'BEGIN' NUMBER←OCTALNO;'IF' T2<>8, 'THEN' T1←1 'ELSE' 'IF' NUMBER<0 'THEN' 'GOTO' FT
         'ELSE' 'BEGIN' NUMBERSCALE←0;FRAC←OCTALFRAC;T1←2 'END' 'END' GENOCT;

'PROCEDURE' SEMICOM;'FOR' T2←READT2 'WHILE' T2=8( 'DO' 'BEGIN' 'INT' B;
   B←0;'FOR' B←B+('IF' READT2=8( 'THEN' +1 'ELSE' 'IF' T2=8) 'THEN' -1 'ELSE' 0) 'WHILE' B>=0 'DO' ;
'END' SEMICOM;

'PROCEDURE' MOVESTRING('LOC' 'INT' FROM,TO);MOVE('BITS'[4,0]FROM←1,FROM,TO);

'PROCEDURE' MCTIDY;'BEGIN' MCTOP←CYCLE((CYCLE(MCTOP,2)+3)'MASK'(-4),22);
      'IF' MCTOP>=STACKSTART 'THEN' GIVEUP("MACRO STACK OVERFLOW")
'END' MCTIDY;

'PROCEDURE' MCMAKE;'BEGIN' STACK[MCTOP]←MCLIST;STACK[MCTOP+1]←-1;MCLIST←MCTOP;MCTOP←MCLIST+2 'END' MCMAKE;

'PROCEDURE' MCMOVE('LOC' 'INT' FROM);
   'BEGIN' MOVESTRING(FROM,STACK[MCTOP]);MCTOP←MCTOP+'BITS'[4,0]FROM←1;STACK[MCLIST+1]←STACK[MCLIST+1]+1 'END' MCMOVE;

'INT' 'PROC' MCLOOK;'BEGIN' MAC←MCLIST;'IF' MAC<>0 'THEN'
   'FOR' MAC←MAC,CHAIN[MAC] 'WHILE' MAC<>0 'DO' 'IF' TESTSTRING(MAC+2,21)=0 'THEN' 'GOTO' ANS;
ANS;'ANSWER' MAC 'END' MCLOOK;

'PROCEDURE' IDENT;
      'BEGIN' MCID←#202020;'FOR' T1←T1,READT2 'WHILE' T2<10 'OR' T2>=$A 'AND' T2<=$Z 'DO' ADDCHAR 'END' ;DENT;

'PROCEDURE' MCIDENT;'IF' READT1>=$A 'AND' T1<=$Z 'OR' T1=$E 'OR' T1=$X 'THEN' ;DENT 'ELSE' 'GOTO' ME;

'INT' 'PROC' MCCHAR('VAL' 'INT' CHAR);'CODE' 'BEGIN' 'SPECIAL' 'ARRAY' S←24,18,12,6;
   1LDXMCTOP;2LDXMCTOP;2SRL22;2LDXS[@2];6LDXSTACK[@1];6SRL@2;5LDX@7;
   5SRL6;6SLL@2;6STOSTACK[@1];1ADD#20000000;1ADD[3];1STOMCTOP;0JCS84
'END' MCCHAR;
```



```
'PAGE' READER 3;

T←T1; SP;T1←'IF' T2>=0 'THEN' T2 'ELSE' READ; 'GOTO' CHARACTER[T1+1];

( HALT DIRECTIVE )

HA;HALT("DIRECTIVE");READT2;'GOTO' SP;

( MACRO FORMAT ERRORS )

MF;GIVEUP("MACRO ERROR");

( DEFINITION OF MACROS )

DF;MCFLAG←1;MCMAKE;MCIDENT;MCMOVE(MCID);'IF' T2=$( 'THEN'
MP;'BEGIN' MCIDENT;MCMOVE(MCID);'IF' T2=$, 'THEN' 'GOTO' MP;'IF' T2<>$) 'THEN' 'GOTO' ME;READT2 'END' ;
   'IF' T2<>$" 'THEN' 'GOTO' ME;
MB;'IF' MCCHAR(READ)<>$" 'OR' MCCHAR(READT2)=$" 'THEN' 'GOTO' MB;'IF' T2<>$; 'THEN' 'GOTO' ME;
   MCFLAG←0;MCTIDY;SEMICOM;'GOTO' SP;

( DELETION OF MACROS )

DL;MCIDENT;'IF' MCLOOK=0 'OR' T2<>$; 'THEN' 'GOTO' ME;
   'IF' MAC=MCLIST 'AND' MCCHAIN=0 'THEN' 'BEGIN' MCLIST←CHAIN[MCLIST];MCTOP←MAC 'END' 'ELSE' STACK[MAC+2]←0;
   SEMICOM;'GOTO' SP;

( " STRINGS " )

ST;MCID←#2U2020;'FOR' T1←READ 'WHILE' T1<>$" 'OR' READT2=$" 'DO' ADDCHAR;MOVESTRING(MCID,NAME);T1←5;'GOTO' EX;

( BRACKETED COMMENT AFTER ; )

SC;SEMICOM;'GOTO' EX;

( IDENTIFIERS AND MACRO EXPANSION )

IN;IDENT;'IF' T=8 'THEN' 'GOTO' SP;( COMMENT AFTER 'END' )
   'IF' MCLOOK=0 'THEN' 'BEGIN' MOVESTRING(MCID,NAME);T1←$N;'GOTO' EX 'END' IDENTIFIER NOT MACRO;
   'BEGIN' 'INT' CTR,LEV;CTR←STACK[MAC+1];
      'IF' CTR<>0 'THEN' 'IF' T2<>$( 'THEN' 'GOTO' ME 'ELSE' T2←-1 'ELSE' 'IF' T2=$| 'THEN' T2←-1;
      MOVE(6,MCCHAIN,STACK[MCTOP]);MCCHAIN←MCTOP;MCTOP←MCCHAIN+6;MCBODY←'BITS'[4,0]MCID+MAC+3;MCFLAG←2;MCTIDY;
      'IF' CTR<>0 'THEN' 'BEGIN'
MM:      MCMAKE;MCMOVE(STACK[MCRODY]);MCBODY←MCBODY+18;T8'[4,0]STACK[MCBODY]←1;LEV←0;
         'FOR' T2←READ 'WHILE' LEV<>0 'OR' T2<>$, 'AND' T2<>$) 'DO'
            'IF' MCCHAR(T2)=$" 'THEN'
               'BEGIN' MCCHAR($");'FOR' T2←MCCHAR(READ) 'WHILE' T2<>$" 'DO';MCCHAR($") 'END'
            'ELSE' 'IF' T2=$$ 'THEN' 'BEGIN' 'IF' MCCHAR(READ)=$" 'THEN' MCCHAR($") 'END'
            'ELSE' LEV←LEV+('IF' T2=$( 'OR' T2=$[ 'THEN' +1 'ELSE' 'IF' T2=$) 'OR' T2=$] 'THEN' -1 'ELSE' 0);
         MCCHAR($");MCCHAR($,);MCTIDY;CTR←CTR-1;'IF' CTR<>0 'THEN' 'IF' T2=$, 'THEN' 'GOTO' MM 'ELSE' 'GOTO' ME;
         'IF' T2<>$) 'THEN' 'GOTO' ME;STACK[MCCHAIN+2]←MCSOURCE
      'END' PARAMETERS;
      MCSOURCE←MCBODY+'LOC'(STACK[0]);T2←-1;MCFLAG←0;'GOTO' SP
   'END' MACRO EXPANSION;

( HEXADECIMAL NUMBERS )

HX;NUMBER←0;
HV;'IF' READT2>=10 'THEN' 'IF' T2<$A 'OR' T2>$F 'THEN' 'GOTO' INTX 'ELSE' T2←T2-23;NUMBER←CYCLE(NUMBER,4)+T2;'GOTO' HV;
```

```
'PAGE' READER 4;

( : UNDERLINED WORDS : )

PR;READT2;T1←RECOG(1);'IF' T1<72 'THEN' 'GOTO' OK 'ELSE' 'GOTO' ULWORD[T1-71];

( OPTIONAL ITEMS; IGNORE FROM ? TO ; IF KEY1=1 )

QM;'IF' 'BIT'[1][4]<>0 'THEN' 'GOTO' CH;READT2;'GOTO' SP;

( FROM 'COMMENT' AND 'PAGE' TO ; )

PG;'IF' LEVEL<>0 'THEN' 'BEGIN' NEWLINE;TEXT("PAGE;") 'END' 'ELSE' T1←0;
CM;'FOR' T2←READ 'WHILE' T2<>$; ,64,64 'DO' 'IF' T1=83 'THEN' OUTCHAR(T2);SEMICOM;'GOTO' SP;

( TEST FOR := )

CH;'IF' READT2<>$= 'THEN' 'GOTO' EX 'ELSE' T1←$+;'GOTO' OK;

( TEST FOR ** )

AS;'IF' READT2<>$* 'THEN' 'GOTO' EX 'ELSE' T1←62; 'GOTO' OK;

( TEST FOR <= AND <> )

LS;T1←4; 'IF' READT2=$= 'THEN' 'BEGIN' RELOP←-1;'GOTO' OK 'END'
     'ELSE' 'IF' T2=$> 'THEN' 'BEGIN' RELOP←0;'GOTO' OK 'END' 'ELSE' RELOP←2;'GOTO' EX;

ES;T1←4; RELOP←1;'GOTO' OK;

( TEST FOR >= )

GS;T1←4; 'IF' READT2<>$= 'THEN' 'BEGIN' RELOP←-2;'GOTO' EX 'END' 'ELSE' RELOP←3;'GOTO' OK;

( UNDERLINED RELATIONAL OPERATORS )

RO;RELOP←T1-74;T1←4;'GOTO' OK;

( DECIMAL NUMBERS AND XF OF CODE )

DD;'IF' READT2>=$A 'AND' T2<=$Z 'AND' T1<8 'THEN' 'BEGIN' ACCUMULATOR←T1;FUNCTION←RECOG(0);T1←6;'GOTO' EX 'END' XF;
   NUMBERSCALE←0;FRAC←0;NUMBER←NOASSY(10,7);'IF' NUMBER<0 'THEN' 'GOTO' FT;'IF' T2=$E 'THEN' 'GOTO' EXPT;
   'IF' T2<>$. 'THEN' 'GOTO' INTX 'ELSE' READT1;READT2;NOASSY(10,6);
   'CODE' 'BEGIN' 'SPECIAL' 'ARRAY' X←5,50,500,5000,50000,500000;
          0STO[2];1LDXNODS;7DIVX[81-T1];7LDX[2];7STOFRAC 'END';
   'IF' T2<>$B 'THEN' 'GOTO' NORM;
```



```
'PAGE' READER 5;

EXPT;'BEGIN' 'INT' A,E;'IF' READT2=$- 'THEN' 'BEGIN' READT2;A←#31463146;(0,8) E←-3 'END'
     'ELSE' 'BEGIN' 'IF' T2=$+ 'THEN' READT2;A←#24000000;(0,625) E←4 'END' ;
     'IF' T2>=10 'THEN' 'GOTO' FT;'FOR' T2←T2:-1;1 'DO' 'BEGIN' NUMBERSCALE←NUMBERSCALE+E; 'CODE' 'BEGIN'
       7LDXFRAC;7SRL1;7MPYA;7SLL1;6LDXNUMBER;6MPYA;7ADD[2];6ADD[3];7STOFRAC;6STONUMBER 'END' CODE 'END' FOR;
     READT2 'END' DECIMAL EXPONENT;

NORM;'CODE' 'BEGIN' SLDXFRAC;5STO[2];6LDX#4000000;6EXCNUMBERSCALE;6ADD23;7LDXNUMBER;
       7JNZ←3;7SLL23;6SUB23;OOVR←1;7SLV85;6ADD#5;7SRL1;7ADD[1];6ADD1;7JLT←-3;
       7STONUMBER;7JZEZERO;6ADSNUMBERSCALE;6AND#77777700;6JNZFT 'END' NORM CODE;T1←2;'GOTO' EX;

ZERO;NUMBERSCALE←#40000167;T1←0;'GOTO' EX;
INTY;READT2;
INTX;'CODE' 'BEGIN' 7LDXNUMBER;7JZEZERO;0STO[2];OOVR←1;7SLV86;6SLC6;6ADD#3/777167;6STONUMBERSCALE 'END';T1←1;'GOTO' EX;

( EXPONENT PART ONLY )

AM;NUMBER←1024;FRAC←0;NUMBERSCALE←-10;'GOTO' EXPT;

( # OCTAL NUMBERS )

OC;READT1;
OQ;'IF' T1=$J 'THEN' 'BEGIN' NUMBER←OCTALFRAC;'GOTO' INTX 'END' LEFT JUSTIFIED 'ELSE' GENOCT;
OX;'IF' T1=1 'THEN' 'GOTO' INTX 'ELSE' 'GOTO' NORM;

( 'OCTAL' NUMBERS )

OT;'IF' READT1<>$( 'THEN' 'GOTO' OQ 'ELSE' READT1;GENOCT;'IF' T2<>$) 'THEN' 'GOTO' FT 'ELSE' READT2;'GOTO' OX;

( 'LITERALS' )

LT;'IF' READT2<5 'THEN' 'BEGIN' NODS←T2;READT2 'END' 'ELSE' NODS←1;NUMBER←0;'IF' T2<>$( 'THEN' 'GOTO' FT;
   'FOR' NODS←NODS;-1;1 'DO' NUMBER←CYCLE(NUMBER,6)+READ;'IF' READ=$) 'THEN' 'GOTO' INTY 'ELSE' 'GOTO' FT;

( $ CONSTANTS )

CH;NUMBER←READ;'GOTO' INTY;

( ACCUMULATORS )

AC;'IF' READT2>=8 'THEN' 'GOTO' FT 'ELSE' NACC←T2+AMARK;'GOTO' OK;

( FINISH )

F;;T1←48;T2←63;'GOTO' EX;

( CHARACTER FAULTS )

FT;GIVEUP("CHARACTER FAULT");

( SHIFT OPERATORS )

SH;FUNCTION←T1-54;T1←3;

( COMMON EXIT TO CLEAR T2 )

OK;T2←-1;

( EXIT IF T2 SET )

EX;'END' READER;'GOTO' EXIT;
```

```
'PAGE' TRACE AND STATUS;


ZERONUM:NUMBER=0;'GOTO' EXIT;

NEGNUM:NUMBER:=-NUMBER;'GOTO' EXIT;

SFTTEN:TOPTEN:=NUMBER;'GOTO' EXIT;

SFTYES:ACCUMULATOR:=1;'GOTO' EXIT;

SFTNO:ACCUMULATOR:=0;'GOTO' EXIT;

SFTTEST:TEST:=1;'GOTO' EXIT;

SFTLEVEL:LEVEL:=NUMBER;'GOTO' EXIT;

LABTRACE:'BIT'[0]TRACE:=ACCUMULATOR;'GOTO' EXIT;

PROCTRACE:'BIT'[1]TRACE:=ACCUMULATOR;'GOTO' EXIT;

FORTRACE:'BIT'[2]TRACE:=ACCUMULATOR;'GOTO' EXIT;

ASSTRACE:'BIT'[3]TRACE:=ACCUMULATOR;'GOTO' EXIT;

SETLAB:STATUSCODE:SETLABEL(LOOKUPLAB);'GOTO' EXIT;

GOTOL:'BEGIN' 'SWITCH' S:=S0,S1,S2,S3,S4;'GOTO' S[STATUS+1];
     S4:REVCJ;OUTXFMN(ACCUMULATOR,FUNCTION,0,USELAB(LOOKUPLAB),'LOC'(NAME));STATUS:=5;'GOTO' S5;
     S3:STATUS:=0;'GOTO' S5;
     S2:'IF' DUECHAIN<>0 'THEN' 'GOTO' S5 'ELSE' ANSLINK;
     S1:'IF' DUFCHAIN<>0 'THEN' 'GOTO' S5 'ELSE' DUEERR;
     S0:OUTI(#100,USELAB(LOOKUPLAB),'LOC'(NAME));STATUS:=1;
     S5:JOINCHAINS(DUECHAIN,PCH[LOOKUPLAB]);DUECHAIN:=0
'END' GOTOL;'GOTO' EXIT;

IFS:STATUSCHECK;STATUS:=4;ONSTACK(2,IFCHAIN);SKIPCHAIN:=0;'GOTO' EXIT;

FLSFS:REVCHAIN;'IF' STATUS=4 'THEN' REVCJ 'ELSE' 'IF' STATUS=5 'THEN' STATUS:=0 'ELSE' 'IF' STATUS=0 'THEN' STATUS:=3;
      'GOTO' EXIT;

F1:'IF' STATUS=4 'AND' FUNCTION=#24 'THEN' OUTCJ;'IF' STATUS>=3 'THEN' STATUS:=0;
         JOINCHAINS(SKIPCHAIN,DUECHAIN);OFFSTACK(2,IFCHAIN);'GOTO' EXIT;

OMACT:REVCHAIN;REVCJ;OUTCJ;SETDUE;REVCHAIN;'GOTO' EXIT;

OVRTEST:FUNCTION:=#24;'GOTO' EXIT;
```



```
'PAGE' SEGMENTS;


ENDPROG;'BEGIN' 'INTEGER' DECS;DECS:=LOCALLIMIT;
'IF' STATUS<>1 'OR' DUECHAIN<>0 'THEN' 'BEGIN' SETDUE;CALLLIB(1,"(END)") 'END' PROG;
'FOR' DECS:=DECS 'WHILE' DECS<>0 'DO' 'BEGIN' 'INT' TEST,CTEST;CTEST:='BITS'(3,6)DIRINDADD(DECS)-4;TEST:=TYP[DECS]-LABEL;
     'IF' TEST=0 'AND' AYM[DECS]=0 'THEN'
          'BEGIN' 'INT' LAB;LAB:=SCANLAB(DECS);'IF' LAB<>0 'THEN' 'IF' PCH[LAB]>=0 'THEN'
               'BEGIN' SETLABEL(LAB);OUT2F(DIRADD[DECS],"(EXTLAB)") 'END' 'ELSE' 'IF' CTEST=0 'THEN'
               'BEGIN' OUTCONT(PCH[LAB]);DOSTRING('LOC'(LABID[LAB]),OUTCONT);OUTS(23,DIRADD[DECS]) 'END'
          'END' CASE OF LABEL
     'ELSE' 'IF' TEST>=0 'AND' CTEST=0 'THEN'
          'BEGIN' 'INT' REF;REF:=LOOKUPNAME(LOCALLIMIT,DECS);'IF' REF<>0 'AND' TYPEBITS[REF]=TYPEBITS[DECS] 'THEN'
               'BEGIN' OUTCONT(DIRADD[REF]);DOSTRING('LOC'(STRING[REF]),OUTCONT);
                    'IF' TEST=0 'THEN' OUTS(22,INDADD[DECS]) 'ELSE' OUTS(21,DIRADD[DECS]) 'END'
          'END' CASE OF SWITCH AND PROC;
     DECS:=CHAIN(DECS)
'END' FOR DECS;    ENDBLOCK;FINISHSEG 'END' ENDPROG;'GOTO' EXIT;

BEGINPROG;STARTSEG(1);STATUS:=0;DUECHAIN:=0;BEGINBLOCK;'GOTO' EXIT;

COMON:'IF' CSUM<>0 'THEN' GIVEUP("TWO COMMONS");COMMON:=1;STARTSEG(3);'GOTO' EXIT;

COMOFF:'BEGIN' 'INT' PTR,ADD;'IF' CSUM=0 'THEN' CSUM:=-1;
    OUTCONT(CSUM);FINISHSEG;COMMON:=0;THEAD;OUTS(3,CSUM);
    'FOR' PTR:=DECLIST,CHAIN(PTR) 'WHILE' PTR<>0 'DO' 'BEGIN' ADD:=DIRINDADD(PTR);
        'IF' 'BITS'(2,6)ADD=1 'THEN' 'BEGIN'
            ADD:=ADDADD(#00400000,ADD:=('IF' 'BIT'(8)ADD<>0 'THEN' DATAMAX 'ELSE' 0));
            'IF' DIRADD[PTR]<>0 'THEN' DIRADD[PTR]:=ADD 'ELSE' INDADD[PTR]:=ADD;
            'IF' TYP[PTR]>=LABEL 'THEN'
                'BEGIN' DOSTRING('LOC'(STRING(PTR)),OUTCONT);OUTS(56,ADD) 'END' OUTPUT
            'END' COMMON IDENT
        'END' FOR PTR;
    ENDSEG
'END' COMOFF;'GOTO' EXIT;

SFTLIBVAR;'IF' NUMBER 'MASK' #77740000 = MARK 'THEN' STOPOP('LOC'(NAME),"IS OUT OF RANGE");
    PRINTADD(DIRADD[DECLIST]);TEXT('LOC'(NAME));TEXT(" = #");OCTLOOP(NUMBER,8);NEWLINE;
    THEAD;OUTCONT(DIRADD[DECLIST]);OUTCONT(NUMBER);DOSTRING('LOC'(NAME),OUTCONT);OUTS(6,TSUM);TTAIL;'GOTO' EXIT;

LOOKUPD:LOOKUP;DIRADD[RN]:=DIRINDADD(RN);INDADD[RN]:=0;'GOTO' EXIT;

NEWNAME:'IF' LOOKUPNAME(LOCALLIMIT,0)<>0 'THEN' STOPOP('LOC'(NAME),"DECLARED TWICE");
      'IF' TYP<0 'THEN' XSPARAMS(PROCPTR);
      ONSTACK(6+WDS[0],DECLIST);DIRADD[0]:=ADDADD(DIRADD[0],1);BITSPEC:=0;INDADD[0]:=0;
      'IF' LCM[0]<>0 'THEN' 'BEGIN' LCM[DECLIST]:=0;SWOP(DIRADD[DECLIST],INDADD[DECLIST]) 'END' ;
      'IF' PAC[0]<>0 'THEN' 'BEGIN' NEXTPARAM;'IF' TYP[0]=LABEL 'AND' AYM[0]=0 'THEN' INDADD[DECLIST]:=LOOKUPLAB 'END' ;
'GOTO' EXIT;
```

```
'PAGE' DECLARATIONS;

LABL:GRABVAR;DIRADD[RN]←LOOKUPLAB;TYPEBITS[RN]←#10200067;SPIEL[RN]←'LOC'(LABID[DIRADD[RN]]);'GOTO' EXIT;

LABBK:'IF' TYP[RN]<>LABEL 'THEN' STOPOP(SPIEL[RN],"IS NOT A SWITCH");TYPEBITS[RN]←#00000067;'GOTO' EXIT;

STRINGEX:GRABVAR;SPIEL[RN]←"(STRING)";TYPEBITS[RN]←#10000067;DIRADD[RN]←SPECSTRING;'GOTO' EXIT;

SKIPDTA:'IF' PRESETOK<>0 'AND' DTA<>DATASTART 'THEN' 'BEGIN' DTA←DATASTART;OUTS(31,DTA) 'END' SKIPDTA;'GOTO' EXIT;

STARTDEC:IDTYPE←0;BITSPEC←0;DATASTART←DATAMAX;DIRADD[0]←DATASTART;INDADD[0]←0;'GOTO' EXIT;

DECSIZE;'IF' OVERLAY=0 'THEN' DATAMAX←DIRADD[0];'GOTO' EXIT;

OVERON;OVERBASE←TOSADD;DIRADD[0]←OVERBASE;OVERLAY←1;'GOTO' EXIT;

OVEROFF:OVERLAY←0;'GOTO' EXIT;

TABLESIZE:'IF' OVERLAY=0 'THEN' DATAMAX←ADDADD(DATAMAX,NUMBER);'GOTO' EXIT;

CLEARTYPE:IDTYPE←0;'GOTO' EXIT;

TYPEINT;PARTINT;IDTYPE←IDTYPE←#67;'GOTO' EXIT;

TYPEFLOAT:  IDTYPE←IDTYPE←#00010000;'GOTO' EXIT;
TYPELAB:    IDTYPE←IDTYPE←#00014000;'GOTO' EXIT;
TYPEPROC:   IDTYPE←IDTYPE←#00020000;'GOTO' EXIT;
TYPETPROC:  IDTYPE←IDTYPE←#00024000;'GOTO' EXIT;
TYPEVPROC:  IDTYPE←IDTYPE←#00164000;'GOTO' EXIT;
TYPESWITCH: IDTYPE←IDTYPE←#20014000;'GOTO' EXIT;
TYPELOC:    IDTYPE←IDTYPE←#10400000;'GOTO' EXIT;
TYPESPEC:   IDTYPE←IDTYPE←#00040067;'GOTO' EXIT;
TYPEARRAY:  IDTYPE←IDTYPE←#20000000;'GOTO' EXIT;
TYPEIARRAY: IDTYPE←IDTYPE←#30000000;'GOTO' EXIT;
TYPEISWITCH;IDTYPE←IDTYPE←#30014000;'GOTO' EXIT;

NOBITS;'IF' NUMBER=0 'OR' NUMBER>=25 'THEN' 'GOTO' BITDECFAIL 'ELSE' IDTYPE←IDTYPE←NUMBER←64;'GOTO' EXIT;

NOAFTER;'BEGIN' 'INT' P;P←'BITS'[5,13]IDTYPE←#37-NUMBER;IDTYPE←IDTYPE←P←#00004000;
                        'IF' P<0 'OR' P>=#100 'THEN' 'GOTO' BITDECFAIL 'END' NOAFTER;'GOTO' EXIT;

ENDDECS;DATAPTR←DATAMAX;DATASTART←DATAPTR;PRESETOK←0;'GOTO' EXIT;

ENDST;DATAPTR←DATASTART;'GOTO' EXIT;
```



```
'PAGE' TABLES AND SPECIAL;

PARAMTAB:SWOP(IDTYPE,IDTYPE2);SWOP(DIRADD[0],PARAMPTR2);'GOTO' EXIT;

NOSIG:BITSPEC←24-NUMBER;'GOTO' EXIT;

FIELDPOSN;BITSPEC←(BITSPEC-NUMBER)←32+NUMBER;'IF' BITSPEC>=0 'THEN' 'GOTO' EXIT;

BITDECFAIL:STOPOP("BITS/SCALE","OUT OF RANGE");'GOTO' EXIT;

UNSFIELD;'IF' BITSPEC=0 'THEN' 'GOTO' BITDECFAIL 'ELSE' 'BEGIN' BITSPEC←BITSPEC←MARK;IDTYPE←IDTYPE←64;
      'IF' TYP[0]<>INTEGER 'THEN' 'BEGIN' IDTYPE←IDTYPE←4;'IF' PVL[0]=0 'THEN' 'GOTO' BITDECFAIL 'END'
'END' UNSFIELD;'GOTO' EXIT;

TABADD:TABLED←DIRADD[DECLIST] 'UNION' #10000000:TABLE1←INDADD[DECLIST];'GOTO' EXIT;

FIELDDISP:DIRADD[0]←ADDADD(TABLED,NUMBER);INDADD[0]←TABLE1;'GOTO' EXIT;

OUTPRESET;OUTPRESETO(SCALECON,0);'GOTO' EXIT;

PRESETSTRING;OUTPRESETO('BITS'[14,10]SPECSTRING,STA);'GOTO' EXIT;

ABSADD;DIRADD[0]←NUMBER 'MASK' #37777;'GOTO' EXIT;

LIBADD;DIRADD[0]←NUMBER 'MASK' #37777 ←#00500000;'GOTO' EXIT;

EXTADD;DIRADD[0]←(TOPTEN←#1000←(NUMBER←#600) 'MASK'#777) 'MASK' #37777←#00600000;'GOTO' EXIT;

SPECCON;OUT1014CS(TOSADD);'GOTO' EXIT;

SPECREL;OUT1014CS(ADDADD(STA,NUMBER));'GOTO' EXIT;

SPECNUM;OUT24CS(SCALECON);'GOTO' EXIT;

SPECLAB;'BEGIN' 'INTEGER' PTR;ADD;'IF' COMMON<>0 'THEN'
      'BEGIN' PTR←LOOKUPNAME(0,0);ADD←0;'IF' PTR=0 'THEN'
            'BEGIN' DIRADD[0]←STA;IDTYPE←#00014000;'GOTO' NEWNAME 'END' FIRST TIME
            'ELSE' 'IF' TYPEBITS[PTR]<>#00014000 'OR' TGD[PTR]<>3 'THEN' STOPOP('LOC'(NAME),"NOT COMMON LABEL")
            'ELSE' 'BEGIN' ADD←DIRADD[PTR];DIRADD[PTR]←STA 'END' CHAINING
      'END' CASE OF LAB IN SPEC ARRAY IN COMMON
'ELSE' 'BEGIN' PTR←LOOKUPLAB;ADD←DCH[PTR];'IF' ADD>=0 'THEN' DCH[PTR]←STA 'END';OUT1014CS(ADD)
'END' SPECLAB;'GOTO' EXIT;

ADDRSW;DIRADD[0]←ADDADD(STA,-1);TOPTEN←0;'GOTO' EXIT;

ADDRSPEC;DIRADD[0]←STA;'GOTO' EXIT;

SPECONE;OUT24CS(0);'GOTO' EXIT;
```

```
'PAGE' PROCEDURES AND ARRAYS;                                                                                    31

BEGINPSPEC;MAKEPSPEC;EXCPSPEC;'IF' SPB[0]=0 'THEN' 'GOTO' EXIT;NEXTPARAM;

NEXTPSET;;DTYPE←;DTYPE 'MASK' #47000000;'GOTO' EXIT;

ENDPSPEC;EXCPSPEC;FINISHPSPEC;'GOTO' EXIT;

ENDPROC;EXCPSPEC;STATUS←1;LOCALLIMIT←PROCPTR;ENDBLOCK;OFFSTACK(8,PROCCHAIN);FINISHPSPEC;'GOTO' EXIT;

BEGINPROC;'IF' STATUS=0 'THEN' 'BEGIN' REVCHAIN;OUTUJ;REVCHAIN 'END' ;
    PROCSTACK;LINK←DIRADD[PROCPTR];DUECHAIN←STA;DIRADD[PROCPTR]←DUECHAIN;OUT24CS(0);
    'IF' 'BIT'[1]TRACE<>0 'THEN' PROCSTRING←SPECSTRING;'GOTO' EXIT;

SETPARAMS;PARAMDECS←DECLIST;DATAMAX←DIRADD[0];'GOTO' EXIT;

KILLPARAMS;DECLIST←PROCPTR;LABDECLIST←0;DATAMAX←LINK;LINK←0;'GOTO' EXIT;

PROCENTRY;'BEGIN' 'INT' P,A;
    'PROC' SCAN('PROC' PROC);
        'BEGIN' P←PARAMDECS;'FOR' P←P 'WHILE' P<>PROCPTR 'DO' 'BEGIN' A←PAC[P];
            'IF' A<>0 'THEN' PROC;P←CHAIN[P] 'END' 'END' SCAN;
    'PROC' LABPARAM;'IF' TYP[P]=LABEL 'AND' AYM[P]=0 'THEN'
        'BEGIN' SETLABEL(INDADD[P]);INDADD[P]←0;OUT27(DIRADD[P],"PARAMLAB") 'END' LABPARAM;
    'PROC' DUMPPARAM;'BEGIN' 'INT' ADD;
        ADD←DIRINDADD(P);ACCS[A]←ADD;OUTX4MN(A,#10,0,ADD,'LOC'(STRING[P])) 'END' DUMPPARAM;

SCAN(LABPARAM);ENTERPROC;OUT1(#1040,LINK,"(LINK)");SCAN(DUMPPARAM);
'IF' PROCSTRING<>0 'THEN' 'BEGIN' ZEROACCS;LIBTRACE(4,PROCSTRING,"(PNAME)") 'END'
'PNB' PROCENTRY;'GOTO' EXIT;

EXITCHECK;'IF' TYP[PROCPTR]=PROCEDURE 'THEN'
        'IF' STATUS<>0 'AND' DUECHAIN=0 'THEN' 'GOTO' EXIT 'ELSE' SETDUE
'ELSE' 'IF' DUECHAIN<>0 'OR' STATUS=0 'OR' EXITCH=0 'AND' STATUS<>2 'THEN'
            STOPOP('LOC'(STRING[PROCPTR]),"EXIT WITHOUT ANSWER")
        'ELSE' 'IF' PROCSTRING<>0 'THEN' 'BEGIN' SETCHAINTOPTR[EXITCH];
            'IF' SPB[PROCPTR]=0 'THEN' OUT1(#1420,STB[PROCPTR]-#24000,"(TYPE)") 'ELSE' OUT1(#1400,LINK+1,"(TYPE)");
            CALLLIB(6,"(PTARS)") 'END'
        'ELSE' 'IF' STATUS=1 'THEN' 'GOTO' EXIT;
'IF' PROCSTRING<>0 'THEN' LIBTRACE(5,PROCSTRING,'LOC'(STRING(PROCPTR)));   OUT27(LINK,"(EXIT)");'GOTO' EXIT;

SETLIBSFG;STAR'SEG(2);PROCSTACK;DUECHAIN←DIRADD[DECLIST]←MARK;LINK←DATASTART;DIRADD[0]←LINK+1;TRACE←0;'GOTO' EXIT;

ZEROARRAYS;AR2AYS←0;TOPTEN←0;'GOTO' EXIT;

SETLB;OFFSET←NUMBER;'GOTO' EXIT;

SETUB;NUMBER←NUMBER+OFFSET+1;
    'IF' NUMBER<1 'THEN' 'BEGIN' NUMBER←1;STOPOP('LOC'(NAME),"ARRAY BOUND ERROR") 'END' ;'GOTO' EXIT;

MIDIM;DODIM(ADDADD(STA,ARRAYS),NUMBER,OUT1014CS);

LASTDIM;DODIM(ARRAYBASE,NUMBER,OUT1014CS);

ONEDIM;ADDRARRAY(ARRAYBASE);

FIRSTDIM;ADDRARRAY(STA);

ENDARRAY;    OVERLAY<>0 'THEN' OVERBASE←ADDADD(OVERBASE,ARRAYS) 'ELSE' DATAMAX←ADDADD(DATAMAX,ARRAYS);'GOTO' EXIT;
```

```
'PAGE' ARITHMETIC 1;                                                                                             32

PLUSSUB;SUBTERM(00);

MINUSSUB;SUBTERM(01);

SUBCOMMA;'BEGIN' 'INT' LH;LH←EXPRCHAIN-5;'IF' INDADD[LH]=0 'THEN' INDADD[LH]←DIRADD[LH]
    'ELSE' 'BEGIN' 'INT' MOD;MOD←GOODONE(LH,3);INST(LH,MOD,00);INDADD[LH]←MOD+AMARK 'END';
    DIRADD[LH]←0;ACCUPDATE(LH) 'END' SUBCOMMA;'GOTO' EXIT;

SETSHIFT;BITSHIFT←FUNCTION(NONAME;SPIEL[RH]←"(SHIFT)";

SETUPSUB;'IF' AYM[RH]<>0 'THEN' AYM[RH]←0 'ELSE' STOPOP(CHAIN[RH],"IS NOT AN ARRAY");
        STACKEXPR;EXPSCALE←#1767;SCALEFIRM←1;PREFACC←FINDACC(3);'GOTO' EXIT;

MSK;SWOPOPT(OPERATE(#15);

NEG;SWOPOPT(OPERATE(#16);

ORF;SWOPOPT(OPERATE(#17));

ANDSHIFT;KILLEXPR;OPERATE(BITSHIFT);

ONEBIT;BITSPEC←23;;IDTYPE←#167;'GOTO' EXIT;

LHSBITS;'IF' PARTWORD[RH]<>0 'OR' TYP[RH]>FLOATING 'THEN' STOPOP(SPIEL[RH],"ILLEGAL BITS ASSIGNMENT");
    STB[RH]←IDTYPF;PARTWORD[RH]←BITSPEC;'GOTO' EXIT;

RHSBITS;'IF' TYPEBITS[RH]<0 'OR' LCM[RH]<>0 'OR' PARTWORD[RH]<>0 'THEN' GOODPICK(RH,0);
    PARTWORD[RH]←BITSHIFT;TYPEBITS[RH]←'BITS'[13,1]BITSHIFT;'GOTO' EXIT;

BITSIN;BITSHIFT←CYCLE(IDTYPE,10)←BITSPEC;'GOTO' EXIT;

TYPEXPR;STACKEXPR;EXPSCALE←IDTYPE;SCALEFIRM←1;

SETPREF;PREFACC←FINDPREF;'GOTO' EXIT;

ANYPREF;PREFACC←FINDACC(7);'GOTO' EXIT;

EXPRTYPE;PICK(RH,GOODACC(RH),EXPSCALE,0);

OFFSTEXPR;OFFSTACK(5,EXPRCHAIN);MOVE(5,STACK[RH],STACK[STACKPOINTER]);
        STACKPOINTER←STACKPOINTER+5;SETLHRH;ACCUPDATE(RH));'GOTO' EXIT;

SCALETERM;'IF' SCALEFIRM=0 'THEN' 'BEGIN' 'INT' SCALE;D;'SWITCH' S←IO,II,IF,FI,FF,FL;
GO;SCALE←STB[RH];'IF' EXPSCALE=0 'THEN' 'GOTO' II 'ELSE' SCALETEST(SCALE,EXPSCALE,S);
IO;EXPSCALE←#67;'GOTO' EX;
II;'IF' EXPSCALE<SCALE 'THEN' EXPSCALE←SCALE;'GOTO' EX;
FL;EXPSCALE←#10000;'GOTO' EX;
FI;SWOP(SCALE,EXPSCALE);
IF;SCALE←#7037←'BITS'[9,13]SCALE;
FF;D←'BITS'[6,18]EXPSCALE-'BITS'[6,18]SCALE;;'IF' D<0 'THEN' 'BEGIN' D←-D;SWOP(SCALE,EXPSCALE) 'END' ;
    D←CYCLE(D,6)-(EXPSCALE-SCALE)'MASK'#77777700;'IF' D<0 'THEN' EXPSCALE←EXPSCALE+D;
EX;'END' FREE SCALE 'ELSE' 'IF' EXPSCALE=#10000 'THEN' FLOATIT(RH)
    'ELSE' 'IF' TYP[RH]=FLOATING 'THEN' 'BEGIN' ACCPICK(RH,7);DUMPACCS;ZEROACCS;
        OUT1(#1420,EXPSCALE,"(TYPE)");CALLLIB(21,"(FIXPP)");
        TYPEBITS[RH]←EXPSCALE;ACCUPDATE(RH)
'PNB' SCALETERM;'GOTO' EXIT;
```

```
'PAGE' ARITHMETIC 2;

ADD;SWOPOPT)ADDSUB(02);

SUB;ADDSUB(03);

RAISE;'IF' TYP[RH]<>INTEGER 'THEN' FLOATOP(5);
      'IF' TYP[LH]=FLOATING 'OR' TYPEBITS[RH]>=0 'OR' DIRADD[RH]<>2 'THEN' FLOATOP(6);
      MOVE(5,STACK[LH],STACK[RH]);'IF' LHACC<>0 'THEN' 'BEGIN' DUMPACC(LHACC);PERM 'END' ACC $0;

MOV;'BEGIN' 'INT' ACC,SCALE;'SWITCH' S:=10,11,IF,F1,FF,FL;
   'IF' TYPEBITS[LH]<0 'THEN' PERM 'ELSE' SWOPOPT;ACC:=GOODACC(LH);SCALETEST(TYPEBITS[LH],TYPEBITS[RH],S);
FL:FLOATOP(1);
IO:SCALE:=#67;'GOTO' IC;
I1:SCALE:=STB[LH]-STB[RH]-#167;'IF' SCALE>=#3067 'THEN' SCALE:=#3067;
IC:PICK(LH,ACC,#67-POWERTWO,0);'IF' POWERTWO=0 'THEN' 'GOTO' MP;TYPEBITS[LH]:=SCALE;'GOTO' EX;
F1;'IF' POWERTWO<>0 'THEN' 'BEGIN' TYPEBITS[LH]:=TYPEBITS[LH]+POWERTWO;'GOTO' E; 'END' ;
   'IF' TYPEBITS[RH]<0 'THEN' 'BEGIN' FIXCON;'GOTO' FF 'END' ;PERM;ACC:=GOODACC(LH);
IF:PICK(LH,ACC,#7037+SGB[LH],0);'GOTO' FC;
FF:PICK(LH,ACC,STB[LH],0);
FC:SCALE:=PVL[LH]+PVL[RH]-#40;
   'IF' 'BITS'[18,0]SCALE<>0 'THEN' WARN("MULT SCALE OUT OF RANGE",0);
   SCALE:=SCALE+('IF' STB[LH]<STB[RH] 'THEN' TYPEBITS[LH] 'ELSE' TYPEBITS[RH])'MASK' #37700;
MP;'IF' PARTWORD[RH]<>0 'OR' LCM[RH]<>0 'THEN' GOODPICK(RH,0);
   INST(RH,ACC,#36)TYPEBITS[LH]:=SCALE;'IF' SCALE<#4000 'THEN'
        'BEGIN' OUTXFMN(ACC,#31,0,23,"($N;F7)");OUTXFMN(0,#32,0,1,"($N;F7)");OUTXFMN(ACC,#02,0,2,"(ADD'0)") 'END' ;NT;
EX;'END' MULTIPLY;'GOTO' BENEADEXIT;

DIVIDE;'BEGIN' 'INT' SCALE,SHIFTS;'SWITCH' S:=10,11,IF,F1,FF,FL;'INT''PROC' ENVTEST;
      'ANSWER' 'IF' T1=$0 'OR' T1=$/ 'OR' SCALEFIRM=0 'AND' 'BITS'[6,18]SCALE>='BITS'[6,18]EXPSCALE 'THEN' 0 'ELSE' 1;
   INACC(LH);'IF' LCM[RH]<>0 'OR' PARTWORD[RH]<>0 'THEN' GOODPICK(RH,0);
   'IF' EXPSCALE<>#10000 'THEN' SCALETEST(TYPEBITS[LH],TYPEBITS[RH],S);
FL:FLOATOP(4);
I1:SCALE:=STB[LH]-STB[RH]-#267;'IF' SCALE<0 'THEN' 'GOTO' IO;
'IF' SCALE>=#3067 'THEN' SCALE:=#3067;'GOTO' IC;
IO:SCALE:=#67;
IC:'IF' POWERTWO=0 'THEN' 'GOTO' DC;
   TYPEBITS[LH]:=TYPEBITS[LH]-POWERTWO;PICK(LH,GOODACC(LH),'IF' ENVTEST<>0 'THEN' EXPSCALE 'ELSE' SCALE,0);'GOTO' EX;
IF:SCALE:=TYPEBITS[RH]'MASK'#37700+#77+SGB[LH]-PVL[RH];'GOTO' DC;
F1;'IF' POWERTWO<>0 'THEN' 'BEGIN' TYPEBITS[LH]:=TYPEBITS[LH]-POWERTWO;'GOTO' EX 'END' ;
   'IF' TYPEBITS[RH]>=0 'THEN' 'BEGIN' SCALE:=STB[LH];'GOTO' DC 'END' ; SMALL;
   FIXCON;
FF:SCALE:=('IF' STB[LH]<STB[RH] 'THEN' TYPEBITS[LH] 'ELSE' TYPEBITS[RH])'MASK'#37700+PVL[LH]-PVL[RH]+#40;
DC:'IF' ENVTEST<>0 'THEN' SCALE:=EXPSCALE;
   SHIFTS:='BITS'[6,18]SCALE:=PVL[RH]-PVL[LH]-#40;'IF' SHIFTS<23 'THEN' OUT;(#40,2,"(CLEAR)");
   'IF' SHIFTS<0 'THEN' PICK(LH,GOODACC(LH),STB[LH]+SHIFTS,0)
   'ELSE' 'BEGIN' GOODPICK(LH,0);'IF' SHIFTS<>0 'THEN' OUTXFMN(LHACC,#30,0,SHIFTS,"(SCALE)") 'END' ;
   INST(RH,LHACC,#37);'IF' RHACC=PREFACC 'THEN' PERM;
   OUTXFMN(LHACC,00,0,2,"(QUOT)");TYPEBITS[LH]:=SCALE;
EX;'END' DIVIDE;'GOTO' BENEADEXIT;
```



```
'PAGE' ANSWER AND FOR;

GOTOSK;INST(RH,4,#27);STATUS:=1;'GOTO' BENEADEXIT;

SETANS;'BEGIN' 'INTEGER' DUMMYBLOCK;PREFACC:=7;'IF' LINK=0 'THEN' 'GOTO' FAULT;
      EXPSCALE:=STB[PROCPTR]-#24000;'IF' EXPSCALE<0 'THEN' 'GOTO' FAULT;
      'IF' SPB[PROCPTR]<>0 'THEN' 'GOTO' WEAK;SCALEFIRM:=1;'GOTO' EXIT;
FAULT:STOPOP("ANSWER","NOT ALLOWED");
WEAK: EXPSCALE:=0;SCALEFIRM:=0;'GOTO' EXIT
'END' SETANS;

ANSCHECK;PICK(RH,7,EXPSCALE,0);STATUS:=2;'GOTO' BENEADEXIT;

STOREZERO;'IF' PARTWORD[RH]=0 'THEN' ASSFUN:=ASSFUN+MARK;

CONSTANT;GRABVAR;SPIEL[RH]:="(CONST)";TYPEBITS[RH]:=NUMBERSCALE;DIRADD[RH]:=NUMBER;'GOTO' EXIT;

ANDA;'IF' LHEQRH<>0 'THEN' ASSFUN:=#12;'GOTO' EXIT;

SUBA;'IF' TYPEBITS[RH]<0 'THEN' UNARYMINUS 'ELSE'
    'BEGIN' 'IF' LHEQRH<>0 'THEN' ASSFUN:=#13;ASSFUN:=ASSFUN+MARK 'END' SUBA;'GOTO' EXIT;

SIMPLEASS;'IF' ASSFUN=#12 'THEN' ASSFUN:=0 'ELSE' 'IF' ASSFUN=MARK+#10 'THEN'
    'IF' TYP[LH]=FLOATING 'OR' TYP[RH]=FLOATING 'OR' PARTWORD[LH]<>0 'OR' PARTWORD[RH]=0 'AND' INDADD[LH]=0
        'AND' COPYINACC(RH)=0 'THEN' 'BEGIN' UNARYMINUS;ASSFUN:=#10 'END' 'ELSE' ASSFUN:=#11;'GOTO' EXIT;

STORE;STOREAWAY('BIT'[3]TRACE);'GOTO' EXIT;

UNILEL;OUTCJ;SETQUE;FORSTATE:=FORSTATE+1;'GOTO' EXIT;

STEPUNT;CCC:=CCC 'HASH' TYPEBITS[RH];'IF' TYPEBITS[RH]<0 'THEN' SCALENUM(RH,EXPSCALE) 'ELSE'
  'IF' STB[RH]<>EXPSCALE 'OR' PARTWORD[RH]<>0 'OR' INDADD[RH]<0 'OR' LCM[RH]<>0 'THEN' PICK(RH,GOODACC(RH),EXPSCALE,0);
  FORSTATE:=FORSTATE+1;

FORSTORE;DATAPTR:=DATASTART;DUMPACCS;DATASTART:=DATAPTR;'IF' RTL<DATASTART 'THEN' RTL:=DATASTART;
        ZEROACCS;BJA:=PTA;'GOTO' EXIT;

STARTFOR;STATUSCHECK;BEGLABBLOCK;ONSTACK(8,FORCHAIN);COPYSKIP:=SKIPCHAIN;
    SKIPCHAIN:=0;RTA:=0;FORSTATE:=0;DATAADD:=DATASTART;RTL:=DATAADD;'GOTO' EXIT;

ASSCV;CCC:=TYPEBITS[RH];STOREAWAY(0);
    'IF' FORSTATE=2 'AND' T1<>SW 'THEN' 'BEGIN' SETRT;FORSTATE:=4 'END' ASSCV;'GOTO' EX;T;

CHECKCV;'IF' TYP[RH]>=FLOATING 'OR' PARTWORD[RH]<>0 'OR' AYM[RH]<>0 'THEN'
    'BEGIN' TYPEBITS[RH]:=67;PARTWORD[RH]:=0;STOPOP(SPIEL[RH],"IS UNSUITABLE") 'END' ;
    EXPSCALE:=STB[RH];SCALEFIRM:=1;PREFACC:=7;ASSFUN:=#10;CV:=RH;DIAG("FOR ",SPIEL[CV]);'GOTO' FORSTORE;

MOREFOR;'IF' FORSTATE=0 'THEN' 'BEGIN' RTA:=PTA;OUTI(#100,0,"(DO)");FORSTATE:=2 'END'
    'ELSE' 'BEGIN' 'IF' FORSTATE<4 'THEN' SETRT;FORCOM;REVCHAIN;SETDUE 'END' ;
    ASSFUN:=#101;BJA:=PTA;ZEROACCS;DATASTART:=DATAADD:=HM1[CV];'GOTO' EXIT;

VARCHECK;'IF' TYP[RH]>FLOATING 'THEN'
            'BEGIN' TYPEBITS[RH]:=67;PARTWORD[RH]:=0;STOPOP(SPIEL[RH],"IS NOT A VARIABLE") 'END' NOT VAR;

AYMCHECK;'IF' AYM[RH]<>0 'THEN' STOPOP(SPIEL[RH],"IS NOT SUBSCRIPTED");'GOTO' EXIT;
```

```
DOCS;'BEGIN' 'SWITCH' S←S0,S0,S2,S3,S4,S5,S5;DATASTART←RTL;'GOTO' S[FORSTATE+1];
S2;'IF' CCC<0 'THEN' FORSTATE←5 'ELSE' FORTEST;'GOTO' S0;
S3;SETCHAINTOPTA(RTA);'GOTO' S0;
S4;FORCOM;OUTUJ;'GOTO' S5;
S5;FORCOM;
S6;SETCHAINTOPTA(RTA);OUTI(#1040,RTL,"(LINK)");DATASTART←DATASTART+1;
S0;DATAPTR←DATASTART;'IF' DATAMAX<DATAPTR 'THEN' DATAMAX←DATAPTR;
   'IF' 'BIT'[2]TRACE<>0 'THEN' 'BEGIN' INST(CV,7,00);OUTI(#1420,EXPSCALE,"(TYPE)");OUTI(#1220,STA,SPIEL[CV]);
      DOSTRING(SPIEL[CV],OUT2&CS);CALLLIB(9,"(TRACE)");ZEROACCS 'END'
'END' DOCS;'GOTO' EXIT;

ENDFOR;'BEGIN' 'INT' A;'SWITCH' S←S0,S1,S2,S1,S4,S5;SETLHRN;'GOTO' S[FORSTATE+1];
S1;'IF' STATUS=0 'THEN' 'BEGIN' UJBACK;STATUS←1 'END' 'ELSE' 'IF'DUECHAIN=0 'THEN' STATUSCHECK;
   'IF' DUECHAIN<>0 'THEN' 'BEGIN' OUTCONT(BJA);OUTS(18,DUECHAIN) 'END' ;'GOTO' S0;
S2;STATUSCHECK;FORINC;UJBACK;STATUS←1;'GOTO' S0;
S4;STATUSCHECK;OUT27(RTL,"(REPT)");STATUS←1;'GOTO' S0;
S5;STATUSCHECK;SUSPIEL;'IF' DIRADD[LH]>=0 'THEN' DIRADD[RH]←DIRADD[RH]+1;
   A←COPYINACC(CV);'IF' A=0 'THEN' 'BEGIN' A←7;INST(CV,7,00) 'END' ;
   INST(LH,A,02);INST(CV,A,#10);'IF' DIRADD[RH]<>0 'THEN' INST(RH,A,03);
   OUTXFMN(A,'IF' DIRADD[LH]>=0 'THEN' #23 'ELSE' #22,0,BJA,"(REPT)");ACCS[A]←0;
S0;DUECHAIN←SKIPCHAIN;SKIPCHAIN←COPYSKIP;DATASTART←DATAADD;OFFSTACK(8,FORCHAIN);ENDLABBLOCK
'END' ENDFOR;'GOTO' EXIT;

SETNEZ;RELOP←0;'GOTO' EXIT;

ZEROCOMP;'IF' RELOP<0 'THEN' UNARYMINUS;'GOTO' EXIT;

CONDADD;'IF' RELOP>=0 'THEN' 'GOTO' SUB 'ELSE' 'GOTO' ADD;

CONDSUB;'IF' RELOP>=0 'THEN' 'GOTO' ADD 'ELSE' 'GOTO' SUB;

CONDPLUS;'IF' RELOP<0 'THEN' PERM;'GOTO' SUB;

CONDMINUS;'IF' RELOP>=0 'THEN' 'GOTO' ADD;UNARYMINUS;PERM;'GOTO' SUB;

RELATION;'BEGIN' 'INT' ADDR,MASK;ADDR←
   'IF' TYPEBITS[RH]>=0 'AND' LCM[RH]=0 'AND' PARTWORD[RH]=0 'AND' DIRADD[RH]>=0 'AND' INDADD[RH]=0
         'THEN' DIRADD[RH] 'ELSE' 0;
   FUNCTION←#20+RELOP 'MASK' 3;
   MASK←'IF' PARTWORD[RH]<>0 'AND' FUNCTION<#22 'OR' PARTWORD[RH]<0 'AND' MSS[RH]<>0 'THEN' FORMMASK(RH) 'ELSE' 0;
   'IF' MASK<>0 'THEN' PARTWORD[RH]←0;GODOPICK(RH,0);ACCUMULATOR←RHACC;
   'IF' MASK<>0 'THEN' OUTXFMN(ACCUMULATOR,#15,0,OUTWCONST(MASK),"(MASK)");KILLEXPR;ACCS[ACCUMULATOR]←ADDR
'END' RELATION;'GOTO' EXIT;
```



```
IFEX;DUMPACCS;COPYDUE←DUECHAIN;ONSTACK(4,IFCHAIN);DUECHAIN←0;SKIPCHAIN←0;'GOTO' EXIT;

THENEX;OUTCJ;SETDUE;'GOTO' EXIT;

ELSEX;OUTUJ;SETDUE;SCALEFIRM←1;'GOTO' BEHEADEXIT;

FIEX;SETDUE;OFFSTACK(4,IFCHAIN);DUECHAIN←COPYDUE;
      GRABVAR;TYPEBITS[RH]←EXPSCALE;SPIEL[RH]←"(IFEX)";TOPACC(PREFACC);'GOTO' EXIT;

ELSEFI;PICK(RH,PREFACC,EXPSCALE,0);REVCHAIN;'GOTO' EXIT;

LHSPROC;SCALEFIRM←0;'IF' TYP[RH]<PROCEDURE 'THEN'
      'BEGIN' STOPOP('LOC'(NAME),"IS NOT A PROCEDURE");PARAMSPEC[RH]←'IF' T1=$( 'THEN' 37 'ELSE' 38 'END' LHSPROC;
'GOTO' EXIT;

FINISHPROC;OFFSTACK(8,EXPRCHAIN);SETLHRN;PARAMSPEC[RH]←0;TOPACC(7);'GOTO' EXIT;

CALLPROC;'BEGIN' 'INT' A,AA;
   'FOR' LH←FPP-5;RH '00' 'BEGIN' A←PAC[LH];
      'FOR' AA←ACCS[A] 'WHILE' AA<0 'AND' AA←MARK←3<>LH 'DO' 'BEGIN'AA←PAC[AA←3];
         OUTXFMN(AA,#14,0,A←AMARK,0);SWOP(ACCS[A],ACCS[AA]);STACK[ACCS[AA]]←AA←AMARK
      'END' SWOPPING;
      'IF' AA<0 'THEN' DIRADD[LH]←A←AMARK;  PICK(LH,A,TYPEBITS[LH],0)
   'END' FOR PARAMETERS;
   OUT27(DIRADD[PNP],SPIEL[PNP]);ZEROACCS
'END' CALLPROC;'GOTO' EXIT;

MULTICALL;STACKPOINTER←FPP;PSP←PARAMSPEC[PNP]-1;'IF' TYPEBITS[PNP]>=0 'THEN'
   'BEGIN' 'IF' PNP>=0 'THEN' 'IF' TYPEBITS[PNP]←#07000000<>TYPEBITS[PSP] 'THEN' 'GOTO' FAULT 'ELSE' MAKEPARAM(0)
      'ELSE' 'BEGIN' MAKEPARAM(1);'IF' TYPEBITS[PSP]<>#06040067 'THEN' 'GOTO' FAULT;MAKEPARAM(0);MAKEPARAM(1) 'END';
      'IF' TYPEBITS[PSP]<0 'THEN' FAULT;'BEGIN' STOPOP(SPIEL[PNP],"INCORRECT MULTIPLE CALL");T1←$) 'END'
   'END' MULTICALL;'GOTO' EXIT;

CODPLAB;FIXLABEL(LOOKUPLAB);'GOTO' EXIT;

CODPSHIFT;NONAME;SPIEL[RH]←"(SHIFT)";

INCTOS;DIRADD[RH]←ADDADD(DIRADD[RH],NUMBER);'GOTO' EXIT;

RELADD;NONAME;SPIEL[RH]←"(SELF)";DIRADD[RH]←ADDADD(PTA,NUMBER);'GOTO' EXIT;

NISACC;NONAME;DIRADD[RH]←NACC;'GOTO' EXIT;

NISMOD;'IF' (NACC-1)*MASK*7>=3 'THEN' STOPOP("ACCUMULATOR","IS NOT MODIFIER") 'ELSE' INDADD[RH]←NACC;'GOTO' EXIT;

OUTCODE;LCM[RH]←0;INST(RH,ACCUMULATOR,FUNCTION);

BEHEADEXIT;BEHEAD;'GOTO' EXIT;

'END' COMPILER

'FINISH'
```

```
A               ACCUMULATORS(26),STACK AND PROCS(3),SUBSCRIPTS AND SCALING(17),ASSIGNMENT AND FOR(4),READER 5(5),
                PROCEDURES AND ARRAYS(5),DO AND IF(9),CALLS AND CODE(7),76;
AA              CALLS AND CODE(11),11;
AAA             DEFINITIONS(2),2;
ARSADD          SYNTAX 2(1),TABLES AND SPECIAL(1),2;
AC              READER 1(1),READER 5(1),2;
ACC             OUTPUT 3(2),ACCUMULATORS(4),ARRAYS AND EXPRESSIONS(10),ARITHMETIC(6),ASSIGNMENT AND FOR(17),
                ARITHMETIC 2(9),48;
ACCOF           ACCUMULATORS(3),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),5;
ACCPICK         ARITHMETIC(4),ARITHMETIC 1(1),5;
ACCR            GLOBAL(1),ACCUMULATORS(11),STACK AND PROCS(2),ARRAYS AND EXPRESSIONS(3),SUBSCRIPTS AND SCALING(2),
                ARITHMETIC(2),ASSIGNMENT AND FOR(3),PROCEDURES AND ARRAYS(1),DO AND IF(2),CALLS AND CODE(4),31;
ACCUMULATOR     GLOBAL(1),CHAINS AND BLOCKS(3),READER 4(1),TRACE AND STATUS(7),DO AND IF(3),CALLS AND CODE(1),16;
ACCUPDATE       ACCUMULATORS(4),STACK AND PROCS(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC(2),ARITHMETIC 1(5),11;
ACO             STACK(1),1;
ACI             STACK(1),1;
ACT             ARRAYS AND EXPRESSIONS(2),ANALYSER(2),4;
ADD             OUTPUT 1(3),OUTPUT 2(5),OUTPUT 3(4),ACCUMULATORS(5),STACK AND PROCS(3),ARRAYS AND EXPRESSIONS(11),
                ASSIGNMENT AND FOR(3),SYNTAX 2(1),SEGMENTS(9),TABLES AND SPECIAL(6),PROCEDURES AND ARRAYS(4),
                ARITHMETIC 2(1),DO AND IF(3),58;
ADDA            SYNTAX 2(2),ANSWER AND FOR(1),3;
ADDADD          STACK AND PROCS(1),ARRAYS AND EXPRESSIONS(2),SUBSCRIPTS AND SCALING(2),SEGMENTS(2),DECLARATIONS(1),
                TABLES AND SPECIAL(3),PROCEDURES AND ARRAYS(3),CALLS AND CODE(2),16;
ADDCHAR         READER 2(2),READER 3(1),3;
ADDR            DO AND IF(3),3;
ADDRARRAY       ARRAYS AND EXPRESSIONS(1),PROCEDURES AND ARRAYS(2),3;
ADDRSPEC        SYNTAX 1(2),TABLES AND SPECIAL(1),3;
ADDRSW          SYNTAX 1(1),TABLES AND SPECIAL(1),2;
ADDSUB          ARITHMETIC(1),ARITHMETIC 2(2),3;
AM              READER 1(1),READER 5(1),2;
AMARK           DEFINITIONS(1),ACCUMULATORS(2),SUBSCRIPTS AND SCALING(6),ASSIGNMENT AND FOR(1),PROCEDURE CALLS(1),
                READER 5(1),ARITHMETIC 1(1),CALLS AND CODE(3),16;
ANS             OUTPUT 3(3),ACCUMULATORS(5),STACK AND LABELS(6),SUBSCRIPTS AND SCALING(5),READER 1(11),READER 2(8),38;
ANSCHECK        SYNTAX 2(1),ANSWER AND FOR(1),2;
ANSLINK         CHAINS AND BLOCKS(2),TRACE AND STATUS(1),3;
ANYMORE         PROCEDURE CALLS(1),SYNTAX 2(1),2;
ANYPREF         SYNTAX 1(1),ARITHMETIC 1(1),2;
ARRAYBASE       ARRAYS AND EXPRESSIONS(1),PROCEDURES AND ARRAYS(2),3;
ARRAYS          STACK(1),ARRAYS AND EXPRESSIONS(4),PROCEDURES AND ARRAYS(4),9;
AS              READER 1(1),READER 4(1),2;
ASSCV           SYNTAX 2(1),ANSWER AND FOR(1),2;
ASSFUN          GLOBAL(1),ARITHMETIC(6),ASSIGNMENT AND FOR(5),ANSWER AND FOR(13),25;
ASSTRACE        SYNTAX 1(1),SYNTAX 2(1),TRACE AND STATUS(1),3;
AVM             STACK(1),PROCEDURE CALLS(3),SEGMENTS(2),PROCEDURES AND ARRAYS(1),ARITHMETIC 1(2),ANSWER AND FOR(2),11;
AVMCHECK        SYNTAX 1(1),ANSWER AND FOR(1),2;

B               SUBSCRIPTS AND SCALING(6),READER 2(5),11;
BASE            READER 2(5),5;
BBB             DEFINITIONS(2),2;
BEGINBLOCK      CHAINS AND BLOCKS(1),STACK AND PROCS(1),SYNTAX 1(2),SEGMENTS(1),5;
BEGINPROC       SYNTAX 1(2),SYNTAX 2(1),PROCEDURES AND ARRAYS(1),4;
BEGINPROG       SYNTAX 1(1),SEGMENTS(1),2;
BEGINPSPEC      SYNTAX 1(1),PROCEDURES AND ARRAYS(1)/2;
BEGLABBLOCK     STACK AND LABELS(1),CHAINS AND BLOCKS(1),ANSWER AND FOR(1),3;
BEHEAD          STACK AND PROCS(2),SUBSCRIPTS AND SCALING(1),ARITHMETIC(1),ASSIGNMENT AND FOR(3),SYNTAX 1(1),
                SYNTAX 2(1),CALLS AND CODE(1),10;
BEHEADEXIT      SUBSCRIPTS AND SCALING(1),ARITHMETIC(2),ARITHMETIC 2(2),ANSWER AND FOR(2),CALLS AND CODE(2),9;
BITDECFAIL      DECLARATIONS(2),TABLES AND SPECIAL(3),5;
BITSHIFT        GLOBAL(1),ARITHMETIC 1(5),6;
BITSIN          SYNTAX 2(1),ARITHMETIC 1(1),2;
```



```
BITSPEC         STACK(1),INITIALISATION(1),SEGMENTS(1),DECLARATIONS(1),TABLES AND SPECIAL(7),ARITHMETIC 1(3),14;
BJA             GLOBAL(1),ASSIGNMENT AND FOR(1),ANSWER AND FOR(2),DO AND IF(2),6;
BLOCKCHAIN      GLOBAL(1),CHAINS AND BLOCKS(2),3;
BOOLWORD        SYNTAX 2(1),ANALYSER(1),2;

C               OUTPUT 1(5),5;
C0              PROCEDURE CALLS(2),2;
C1              PROCEDURE CALLS(2),2;
C2              PROCEDURE CALLS(2),2;
C3              PROCEDURE CALLS(5),5;
CALLLIB         OUTPUT 3(2),ARITHMETIC(2),ASSIGNMENT AND FOR(1),SEGMENTS(1),PROCEDURES AND ARRAYS(1),ARITHMETIC 1(1),
                DO AND IF(1),9;
CALLPROC        SYNTAX 2(2),CALLS AND CODE(1),3;
CCC             GLOBAL(1),ANSWER AND FOR(3),DO AND IF(1),5;
CH              READER 1(1),READER 5(1),2;
CHAIN           STACK(1),STACK AND LABELS(13),CHAINS AND BLOCKS(1),STACK AND PROCS(1),ARRAYS AND EXPRESSIONS(1),
                READER 2(1),READER 3(1),SEGMENTS(2),PROCEDURES AND ARRAYS(1),ARITHMETIC 1(1),23;
CHAR            OUTPUT 1(1),OUTPUT 2(1),READER 2(1),3;
CHARACTER       READER 1(1),READER 3(1),2;
CHEAT           ANALYSER(2),2;
CHECKCV         SYNTAX 2(1),ANSWER AND FOR(1),2;
CHECKSUM        DEFINITIONS(1),OUTPUT 2(1),READER 1(1),3;
CHOOSE          READER 1(3),3;
CHS             STACK(1),1;
CLASS           PROCEDURE CALLS(2),2;
CLEARTYPE       SYNTAX 1(5),SYNTAX 2(7),DECLARATIONS(1),13;
CM              READER 1(1),READER 4(2),3;
CN              READER 1(1),READER 4(1),2;
CODELAB         SYNTAX 1(1),CALLS AND CODE(1),2;
CODESHIFT       SYNTAX 2(2),CALLS AND CODE(1),3;
COMMON          GLOBAL(1),INITIALISATION(1),READER 1(1),SEGMENTS(2),TABLES AND SPECIAL(1),6;
COMOFF          SYNTAX 1(1),SEGMENTS(1),2;
COMON           SYNTAX 1(1),SEGMENTS(1),2;
COMPILER20      STACK(1),1;
CON             STACK(1),1;
CONDADD         SYNTAX 2(1),DO AND IF(1),2;
CONDMINUS       SYNTAX 2(1),DO AND IF(1),2;
CONDPLUS        SYNTAX 2(2),DO AND IF(1),3;
CONDSUB         SYNTAX 2(1),DO AND IF(1),2;
CONST           OUTPUT 2(2),OUTPUT 3(2),4;
CONSTANT        SYNTAX 1(1),SYNTAX 2(5),ANSWER AND FOR(1),7;
COPYDUE         GLOBAL(1),CALLS AND CODE(2),3;
COPYINACC       ACCUMULATORS(2),ARRAYS AND EXPRESSIONS(1),ASSIGNMENT AND FOR(6),ANSWER AND FOR(1),DO AND IF(1),11;
COPYSKIP        GLOBAL(1),ANSWER AND FOR(1),DO AND IF(1),3;
CSUM            GLOBAL(1),CHAINS AND BLOCKS(1),INITIALISATION(1),READER 1(1),SEGMENTS(5),9;
CTEST           SEGMENTS(6),6;
CTR             READER 3(7),7;
CV              GLOBAL(1),ASSIGNMENT AND FOR(1),ANSWER AND FOR(3),DO AND IF(6),11;
CYCLE           DEFINITIONS(1),OUTPUT 1(2),OUTPUT 2(2),ARITHMETIC(1),ASSIGNMENT AND FOR(1),PROCEDURE CALLS(1),
                READER 1(3),READER 2(3),READER 3(1),READER 5(1),ARITHMETIC 1(2),18;

D               ARITHMETIC 1(9)29;
DAD             STACK(1),1;
DATAADD         GLOBAL(1),ANSWER AND FOR(3),DO AND IF(1),5;
DATAMAX         GLOBAL(1),ACCUMULATORS(2),CHAINS AND BLOCKS(7),STACK AND PROCS(1),ARRAYS AND EXPRESSIONS(1),
                SEGMENTS(1),DECLARATIONS(5),PROCEDURES AND ARRAYS(4),DO AND IF(2),24;
DATAPTR         GLOBAL(1),ACCUMULATORS(6),DECLARATIONS(3),ANSWER AND FOR(2),DO AND IF(3),15;
DATASTART       GLOBAL(1),CHAINS AND BLOCKS(2),DECLARATIONS(6),PROCEDURES AND ARRAYS(1),ANSWER AND FOR(6),
                DO AND IF(5),21;
DC              ARITHMETIC 2(4),4;
```

```
DCH             STACK(1),STACK AND LABELS(5),TABLES AND SPECIAL(2),8;
DD              READER 1(10),READER 4(1),11;
DECLIST         STACK(1),STACK AND LABELS(1),CHAINS AND BLOCKS(3),STACK AND PROCS(5),ARRAYS AND EXPRESSIONS(1),
                INITIALISATION(1),SEGMENTS(8),TABLES AND SPECIAL(2),PROCEDURES AND ARRAYS(3),25;
DECS            STACK AND LABELS(9),CHAINS AND BLOCKS(4),SEGMENTS(17),30;
DECSIZE         SYNTAX 1(1),DECLARATIONS(1),2;
DF              READER 1(1),READER 3(1),2;
DIAG            OUTPUT 3(1),STACK AND LABELS(1),STACK AND PROCS(1),ANSWER AND FOR(1),6;
DIGITS          OUTPUT 1(5),5;
DIGADD          STACK(1),ACCUMULATORS(6),CHAINS AND BLOCKS(3),STACK AND PROCS(7),ARRAYS AND EXPRESSIONS(6),
                SUBSCRIPTS AND SCALING(16),ARITHMETIC(5),ASSIGNMENT AND FOR(11),PROCEDURE CALLS(1),SEGMENTS(12),
                DECLARATIONS(6),TABLES AND SPECIAL(11),PROCEDURES AND ARRAYS(6),ARITHMETIC 1(2),ARITHMETIC 2(1),
                ANSWER AND FOR(1),DO AND IF(7),CALLS AND CODE(6),106;
DIRINDADD       CHAINS AND BLOCKS(1),SEGMENTS(3),PROCEDURES AND ARRAYS(1),5;
DIVIDE          SYNTAX 2(1),ARITHMETIC 2(1),2;
DL              READER 1(1),READER 3(1),2;
DOCS            SYNTAX 2(1),DO AND IF(1),2;
DODIM           ARRAYS AND EXPRESSIONS(2),PROCEDURES AND ARRAYS(2),4;
DOSHIFT         SYNTAX 2(1),ARITHMETIC 1(1),2;
DOSTRING        OUTPUT 3(5),CHAINS AND BLOCKS(1),ASSIGNMENT AND FOR(1),SEGMENTS(4),DO AND IF(1),12;
DTA             GLOBAL(1),CHAINS AND BLOCKS(2),STACK AND PROCS(2),DECLARATIONS(3),8;
DUFCHAIN        GLOBAL(1),CHAINS AND BLOCKS(4),TRACE AND STATUS(5),SEGMENTS(2),PROCEDURES AND ARRAYS(5),DO AND IF(4),
                CALLS AND CODE(3),24;
DUFERR          CHAINS AND BLOCKS(2),TRACE AND STATUS(1),3;
DUMMY           NASTY CODE(1),CHAINS AND BLOCKS(1),2;
DUMMYBLOCK      ANSWER AND FOR(1),1;
DUMPACC         ACCUMULATORS(3),ARITHMETIC(1),ASSIGNMENT AND FOR(1),ARITHMETIC 2(1),6;
DUMPACCS        ACCUMULATORS(1),ARITHMETIC(2),PROCEDURE CALLS(1),ARITHMETIC 1(1),ANSWER AND FOR(1),CALLS AND CODE(1),7;
DUMPPARAM       PROCEDURES AND ARRAYS(2),2;

E               READER 5(4),4;
ELSEFI          SYNTAX 2(2),CALLS AND CODE(1),3;
ELSES           SYNTAX 1(1),TRACE AND STATUS(1),2;
ELSEX           SYNTAX 2(1),CALLS AND CODE(1),2;
ENDARRAY        SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
ENDBLOCK        CHAINS AND BLOCKS(1),SYNTAX 1(2),SEGMENTS(1),PROCEDURES AND ARRAYS(1),5;
ENDDECS         SYNTAX 1(5),DECLARATIONS(1),6;
ENDFOR          SYNTAX 2(1),DO AND IF(1),2;
ENDLABBLOCK     STACK AND LABELS(1),CHAINS AND BLOCKS(1),DO AND IF(1),3;
ENDPROC         SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
ENDPROG         SYNTAX 1(1),SEGMENTS(1),2;
ENDPSPEC        SYNTAX 1(2),PROCEDURES AND ARRAYS(1),3;
ENDSEG          OUTPUT 2(1),CHAINS AND BLOCKS(1),SEGMENTS(1),3;
ENDST           SYNTAX 1(2),DECLARATIONS(1),3;
ENTERPROC       STACK AND PROCS(1),SYNTAX 1(1),PROCEDURES AND ARRAYS(1),3;
EVVTEST         ARITHMETIC 2(3),3;
EX              PROCEDURE CALLS(3),3;
ES              READER 1(1),READER 4(1),2;
EX              READER 3(3),READER 4(5),READER 5(5),ARITHMETIC 1(4),ARITHMETIC 2(6),23;
EXCPSPEC        STACK AND PROCS(2),PROCEDURES AND ARRAYS(3),5;
EXIT            RESCALE(4),ARRAYS AND EXPRESSIONS(1)7SUBSCRIPTS AND SCALING(1),SYNTAX 1(15),SYNTAX 2(12),ANALYSER(1),
                READER 5(1),TRACE AND STATUS(18),SEGMENTS(7),DECLARATIONS(26),TABLES AND SPECIAL(19),PROCEDURES AND ARRAYS(16),
                ARITHMETIC 1(10),ANSWER AND FOR(13),DO AND IF(5),CALLS AND CODE(16),163;
EXITCH          GLOBAL(1),CHAINS AND BLOCKS(3),STACK AND PROCS(1),PROCEDURES AND ARRAYS(2),7;
EXIVCHECK       SYNTAX 1(2),PROCEDURES AND ARRAYS(1)73;
EXPDCHAIN       GLOBAL(1),ARRAYS AND EXPRESSIONS(2),SUBSCRIPTS AND SCALING(1),PROCEDURE CALLS(1),ARITHMETIC 1(2),
                CALLS AND CODE(1),8;
EXPRTYPE        SYNTAX 2(1),ARITHMETIC 1(1),2;
EXPSCALE        GLOBAL(1),ARRAYS AND EXPRESSIONS(1),ARITHMETIC(7),ASSIGNMENT AND FOR(2),PROCEDURE CALLS(11),
                ARITHMETIC 1(18),ARITHMETIC 2(4),ANSWER AND FOR(8),DO AND IF(1),CALLS AND CODE(2),55;
```



```
EXPT            READER 4(1),READER 5(2),3;
EXT1            DEFINITIONS(1),1;
EXT2            DEFINITIONS(1),1;
EXTT            DEFINITIONS(1),1;
EXTADD          SYNTAX 2(1),TABLES AND SPECIAL(1),2;

F               OUTPUT 3(2),ASSIGNMENT AND FOR(5),7;
FAIL            RESCALE(3),SYNTAX 1(30),SYNTAX 2(17),ANALYSER(1),51;
FAULT           ANSWER AND FOR(3),CALLS AND CODE(3),6;
FC              ARITHMETIC 2(2),2;
FF              ARITHMETIC 1(2),ARITHMETIC 2(5),7;
FFLAG           ASSIGNMENT AND FOR(5),5;
FI              SYNTAX 1(1),READER 1(1),READER 5(1),TRACE AND STATUS(1),ARITHMETIC 1(2),ARITHMETIC 2(4),10;
FIELDDISP       SYNTAX 1(2),TABLES AND SPECIAL(1),3;
FIELDPOSN       SYNTAX 1(1),SYNTAX 2(1),TABLES AND SPECIAL(1),3;
FIFX            SYNTAX 2(1),CALLS AND CODE(1),2;
FINDACC         ACCUMULATORS(2),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC(1),ASSIGNMENT AND FOR(1),
                ARITHMETIC 1(2),8;
FINDOUT         ACCUMULATORS(2),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),4;
FINDPREF        ACCUMULATORS(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC 1(1),3;
FINISHPROC      SYNTAX 2(1),CALLS AND CODE(1),2;
FINISHPSPEC     STACK AND PROCS(1),PROCEDURES AND ARRAYS(2),3;
FINISHSEG       CHAINS AND BLOCKS(1),SYNTAX 1(3),SEGMENTS(2),6;
FIRSTDIM        SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
FIXCON          ARITHMETIC(1),ARITHMETIC 2(2),3;
FIXFD           DEFINITIONS(1),ARRAYS AND EXPRESSIONS(1),2;
FIXLABEL        STACK AND LABELS(2),CHAINS AND BLOCKS(1),CALLS AND CODE(1),6;
FL              ARITHMETIC 1(2),ARITHMETIC 2(4),6;
FLOAT           RESCALE(2),2;
FLOATING        DEFINITIONS(1),SUBSCRIPTS AND SCALING(2),ARITHMETIC(2),ARITHMETIC 1(2),ARITHMETIC 2(1),ANSWER AND FOR(6),12;
FLOATIT         ARITHMETIC(3),ARITHMETIC 1(1),4;
FLOATOP         ARITHMETIC(2),ARITHMETIC 2(4),6;
FORCHAIN        GLOBAL(1),ANSWER AND FOR(1),DO AND IF(1),3;
FORCOM          ASSIGNMENT AND FOR(1),ANSWER AND FOR(1),DO AND IF(2),4;
FORINC          ASSIGNMENT AND FOR(2),DO AND IF(1),3;
FORMMASK        OUTPUT 3(2),ASSIGNMENT AND FOR(2),DO AND IF(1),5;
FORSTATE        GLOBAL(1),ASSIGNMENT AND FOR(6),ANSWER AND FOR(10),DO AND IF(3),20;
FORSTORE        ANSWER AND FOR(2),2;
FORTEST         ASSIGNMENT AND FOR(2),DO AND IF(1),3;
FORTRACE        SYNTAX 1(1),SYNTAX 2(1),TRACE AND STATUS(1),3;
FPP             GLOBAL(1),PROCEDURE CALLS(1),CALLS AND CODE(2),4;
FRAC            READER 1(1),READER 2(1),READER 4(2),READER 5(4),8;
FROM            NASTY CODE(1),STACK AND LABELS(4),READER 2(6),11;
FT              READER 1(4),READER 2(4),READER 4(1),READER 5(7),16;
FUN             ARRAYS AND EXPRESSIONS(9),SUBSCRIPTS AND SCALING(2),ARITHMETIC(3),14;
FUNCTION        GLOBAL(1),CHAINS AND BLOCKS(4),PROCEDURE CALLS(4),READER 4(1),READER 5(1),TRACE AND STATUS(3),
                ARITHMETIC 1(1),DO AND IF(2),CALLS AND CODE(1),18;
FUNCTIONS       READER 1(2),2;

GENOCT          READER 2(1),READER 5(2),3;
GIVFUP          OUTPUT 1(1),STACK AND LABELS(1),ANALYSER(2),READER 2(1),READER 3(1),READER 5(1),SEGMENTS(1),8;
GO              ARITHMETIC 1(1),1;
GOODACC         SUBSCRIPTS AND SCALING(2),ARITHMETIC(2),PROCEDURE CALLS(1),ARITHMETIC 1(1),ARITHMETIC 2(4),
                ANSWER AND FOR(1),11;
GOODONE         SUBSCRIPTS AND SCALING(3),ARITHMETIC 1(1),4;
GOODPICK        SUBSCRIPTS AND SCALING(5),PROCEDURE CALLS(1),ARITHMETIC 1(1),ARITHMETIC 2(3),DO AND IF(1),11;
GOTO            ANALYSER(3),3;
GOTOL           SYNTAX 2(1),TRACE AND STATUS(1),2;
GOTOS           SYNTAX 2(1),ARRAYS AND EXPRESSIONS(1),2;
```

```
GRABVAR         STACK AND PROCS(5),PROCEDURE CALLS(1),DECLARATIONS(2),ANSWER AND FOR(1),CALLS AND CODE(1),8;
GS              READER 1(1),READER 4(1),2;

HA              READER 1(1),READER 3(1),2;
HALT            OUTPUT 1(1),INITIALISATION(1),READER 3(1),3;
HMD             STACK(1),1;
HMI             STACK(1),ANSWER AND FOR(1),2;
HX              READER 1(1),READER 3(1),2;
HY              READER 3(2),2;

I               OUTPUT 1(7),OUTPUT 2(2),ACCUMULATORS(4),ARRAYS AND EXPRESSIONS(2),15;
IAD             STACK(1),1;
IC              ARITHMETIC 2(4),4;
ID              READER 1(28),READER 3(1),29;
IDENT           READER 2(2),READER 3(1),3;
IDTYPE          STACK(1),RESCALE(1),STACK AND PROCS(7),SEGMENTS(1),DECLARATIONS(31),TABLES AND SPECIAL(6),
                PROCEDURES AND ARRAYS(2),ARITHMETIC 1(4),52;
IDTYPE2         GLOBAL(1),STACK AND PROCS(2),TABLES AND SPECIAL(1),4;
IF              ARITHMETIC 1(2),ARITHMETIC 2(4),6;
IFCHAIN         GLOBAL(1),TRACE AND STATUS(2),CALLS AND CODE(2),5;
IFEX            SYNTAX 2(1),CALLS AND CODE(1),2;
IFS             SYNTAX 1(1),TRACE AND STATUS(1),2;
II              ARITHMETIC 1(3),ARITHMETIC 2(4),7;
INACC           ACCUMULATORS(3),ARRAYS AND EXPRESSIONS(1),4;
INBUFF          NASTY CODE(2),OUTPUT 1(1),INITIALISATION(1),4;
INC             STACK AND PROCS(2),ARRAYS AND EXPRESSIONS(2),4;
INCTOS          SYNTAX 1(3),SYNTAX 2(1),CALLS AND CODE(1),5;
INCTR           NASTY CODE(3),OUTPUT 1(2),INITIALISATION(2),7;
INDADD          STACK(1),ACCUMULATORS(4),CHAINS AND BLOCKS(1),STACK AND PROCS(5),ARRAYS AND EXPRESSIONS(1),
                SUBSCRIPTS AND SCALING(13),ARITHMETIC(2),ASSIGNMENT AND FOR(5),PROCEDURE CALLS(1),INITIALISATION(1),
                SEGMENTS(6),DECLARATIONS(1),TABLES AND SPECIAL(2),PROCEDURES AND ARRAYS(2),ARITHMETIC 1(3),
                ANSWER AND FOR(2),DO AND IF(1),CALLS AND CODE(1),52;
INST            ARRAYS AND EXPRESSIONS(3),SUBSCRIPTS AND SCALING(2),ARITHMETIC(1),ASSIGNMENT AND FOR(4),ARITHMETIC 1(1),
                ARITHMETIC 2(2),ANSWER AND FOR(1),DO AND IF(5),CALLS AND CODE(1),20;
INSTTYPE        PROCEDURE CALLS(1),SYNTAX 1(1),2;
INTEGER         DEFINITIONS(1),ARRAYS AND EXPRESSIONS(5),SUBSCRIPTS AND SCALING(2),ASSIGNMENT AND FOR(1),TABLES AND SPECIAL(
                ARITHMETIC 2(1),9;
INTPROC         DEFINITIONS(1),PROCEDURE CALLS(1),2;
INTX            READER 3(1),READER 4(1),READER 5(3),5;
INTY            READER 5(3),3;
IO              ARITHMETIC 1(2),ARITHMETIC 2(5),7;
ISINACC         ACCUMULATORS(1);SUBSCRIPTS AND SCALING(3),ARITHMETIC(1),ARITHMETIC 2(1),6;
ISMOD           ARRAYS AND EXPRESSIONS(2),SUBSCRIPTS AND SCALING(1),ASSIGNMENT AND FOR(1),4;

J               READER 1(6),6;
JJ              READER 1(4),4;
JOINCHAINS      STACK AND LABELS(3),TRACE AND STATUS(2),5;

KILLEXPR        ARRAYS AND EXPRESSIONS(1),SYNTAX 1(2),ARITHMETIC 1(1),DO AND IF(1),5;
KILLPARAMS      SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;

L               OUTPUT 1(2),READER 1(2),4;
LAB             STACK AND LABELS(23),SEGMENTS(7),30;
LABCHAIN        GLOBAL(1),STACK AND LABELS(2),3;
LABDECLIST      GLOBAL(1),STACK AND LABELS(6),CHAINS AND BLOCKS(7),INITIALISATION(1),PROCEDURES AND ARRAYS(1),14;
LABEL           DEFINITIONS(1),PROCEDURE CALLS(1),SEGMENTS(3),DECLARATIONS(1),PROCEDURES AND ARRAYS(1),7;
LABID           STACK(1),STACK AND LABELS(3),CHAINS AND BLOCKS(1),SEGMENTS(1),DECLARATIONS(1),7;
LABL            SYNTAX 2(2),DECLARATIONS(1),3;
LABPARAM        PROCEDURES AND ARRAYS(2),2;
LABSK           SYNTAX 2(2),DECLARATIONS(1),3;
```



```
LABSTACKPTR     GLOBAL(1),STACK AND LABELS(8),CHAINS AND BLOCKS(1),INITIALISATION(1),11;
LABTRACE        SYNTAX 1(1),SYNTAX 2(1),TRACE AND STATUS(1),3;
LASTDIM         SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
LBM             STACK(1),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),3;
LCM             STACK(1),ACCUMULATORS(1),STACK AND PROCS(2),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(5),
                PROCEDURE CALLS(2),SEGMENTS(2),ARITHMETIC 1(1),ARITHMETIC 2(2),ANSWER AND FOR(1),DO AND IF(1),
                CALLS AND CODE(1),20;
LEV             READER 3(5),5;
LEVEL           GLOBAL(1),OUTPUT 3(3),STACK AND PROCS(1),INITIALISATION(1),READER 4(1),TRACE AND STATUS(1),8;
LH              GLOBAL(1),ACCUMULATORS(3),STACK AND PROCS(1),SUBSCRIPTS AND SCALING(21),ARITHMETIC(11),ASSIGNMENT AND FOR(40
                ARITHMETIC 1(10),ARITHMETIC 2(39),ANSWER AND FOR(5),DO AND IF(3),CALLS AND CODE(6),138;
LHACC           ACCUMULATORS(1);SUBSCRIPTS AND SCALING(2),ARITHMETIC(1),ARITHMETIC 2(5),9;
LHEORH          ARITHMETIC(1),ANSWER AND FOR(2),3;
LHSBITS         SYNTAX 2(2),ARITHMETIC 1(1),3;
LHSPROC         SYNTAX 2(1),CALLS AND CODE(1),2;
LI              READER 1(1),READER 5(1),2;
LIBADD          SYNTAX 2(1),TABLES AND SPECIAL(1),2;
LIBTRACE        OUTPUT 3(2),PROCEDURES AND ARRAYS(2),4;
LIM             SUBSCRIPTS AND SCALING(4),READER 2(2),6;
LIMIT           STACK AND LABELS(2),2;
LINK            GLOBAL(1),CHAINS AND BLOCKS(1),INITIALISATION(1),PROCEDURES AND ARRAYS(8),ANSWER AND FOR(1),12;
LLS             STACK(1),STACK AND LABELS(1),2;
LM2             STACK(1),STACK AND PROCS(1),2;
LNO             OUTPUT 3(6),6;
LOCACT          STACK AND PROCS(1),PROCEDURE CALLS(1),SYNTAX 2(1),3;
LOCALLIMIT      GLOBAL(1),CHAINS AND BLOCKS(2),STACK AND PROCS(1),INITIALISATION(1),SEGMENTS(3),PROCEDURES AND ARRAYS(1),9;
LOOKUP          STACK AND PROCS(1),SYNTAX 1(1),SYNTAX 2(7),SEGMENTS(1),10;
LOOKUPD         SYNTAX 1(1),SYNTAX 2(1),SEGMENTS(1),3;
LOOKUPLAB       STACK AND LABELS(1),TRACE AND STATUS(4),SEGMENTS(1),DECLARATIONS(1),TABLES AND SPECIAL(1),CALLS AND CODE(1),
LOOKUPNAME      STACK AND LABELS(1),STACK AND PROCS(1),SEGMENTS(2),TABLES AND SPECIAL(1),5;
LR              READER 1(1),READER 4(1),2;
LSS             STACK(1),ARRAYS AND EXPRESSIONS(2),ASSIGNMENT AND FOR(1),4;

M               OUTPUT 1(2),OUTPUT 3(2),READER 1(2),6;
MAC             READER 1(1),READER 2(8),READER 3(5),16;
MAKEPARAM       PROCEDURE CALLS(3),CALLS AND CODE(4),7;
MAKEPSPEC       STACK AND PROCS(2),PROCEDURES AND ARRAYS(1),3;
MARK            DEFINITIONS(1),ACCUMULATORS(2),STACK AND LABELS(3),STACK AND PROCS(4),ARRAYS AND EXPRESSIONS(1),
                SUBSCRIPTS AND SCALING(1),ARITHMETIC(1),PROCEDURE CALLS(2),SEGMENTS(1),TABLES AND SPECIAL(1),
                PROCEDURES AND ARRAYS(1),ANSWER AND FOR(3),CALLS AND CODE(1),21;
MASK            DO AND IF(5),5;
MASKINST        OUTPUT 3(1),ARRAYS AND EXPRESSIONS(1),ASSIGNMENT AND FOR(3),5;
MASTER          STACK AND LABELS(5),5;
MAX             CHAINS AND BLOCKS(4),4;
MB              READER 3(2),2;
MCBODY          GLOBAL(1),INITIALISATION(1),READER 3(6),8;
MCCHAIN         GLOBAL(1),INITIALISATION(1),READER 1(2),READER 3(5),9;
MCCHAR          READER 2(1),READER 3(10),11;
MCFLAG          GLOBAL(1),INITIALISATION(1),READER 1(1),READER 3(4),7;
MCID            STACK(1),READER 2(5),READER 3(6),12;
MCIDENT         READER 2(1),READER 3(3),4;
MCLIST          GLOBAL(1),INITIALISATION(1),READER 2(6),READER 3(3),11;
MCLOOK          READER 2(1),READER 3(2),3;
MCMAKE          READER 2(1),READER 3(2),3;
MCMOVE          READER 2(1),READER 3(3),6;
MCSOURCE        GLOBAL(1),INITIALISATION(1),READER 1(3),READER 3(2),7;
MCSTART         DEFINITIONS(1),INITIALISATION(1),2;
MCTIDY          READER 2(1),READER 3(3),4;
MCTOP           GLOBAL(1),INITIALISATION(1),READER 2(13),READER 3(4),19;
MD              READER 1(1),READER 2(1),READER 3(8),10;
```

```
MIDDIM          SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
MIN             ACCUMULATORS(6),6;
MINF            ARRAYS AND EXPRESSIONS(2),2;
MINUSSUB        SYNTAX 2(1),ARITHMETIC 1(1),2;
MM              READER 3(2),2;
MMD             ARRAYS AND EXPRESSIONS(8),ASSIGNMENT AND FOR(6),ARITHMETIC 1(4),18;
MOREFOR         SYNTAX 2(1),ANSWER AND FOR(1),2;
MOVF            NASTY CODE(1),STACK AND LABELS(3),STACK AND PROCS(1),READER 1(1),READER 2(1),READER 3(1),ARITHMETIC 1(1),
                ARITHMETIC 2(1),10;
MOVESTRING      READER 2(2),READER 3(2),4;
MP              READER 3(2),ARITHMETIC 2(2),4;
MPV             SYNTAX 2(1),ARITHMETIC 2(1),2;
MSK             SYNTAX 2(1),ARITHMETIC 1(1),2;
MSS             STACK(1),ARRAYS AND EXPRESSIONS(4),ASSIGNMENT AND FOR(1),DO AND IF(1),7;
MULTICALL       SYNTAX 2(1),CALLS AND CODE(1),2;

N               NASTY CODE(1),OUTPUT 3(9),STACK AND LABELS(8),STACK AND PROCS(2),READER 1(2),22;
NACC            GLOBAL(1),READER 5(1),CALLS AND CODE(3),5;
NAME            STACK(1),OUTPUT 3(1),CHAINS AND BLOCKS(1),STACK AND PROCS(2),PROCEDURE CALLS(1),READER 3(2),
                TRACE AND STATUS(2),SEGMENTS(4),TABLES AND SPECIAL(1),PROCEDURES AND ARRAYS(1),CALLS AND CODE(1),17;
NFG             ARRAYS AND EXPRESSIONS(4),SUBSCRIPTS AND SCALING(9),13;
NFGNUM          SYNTAX 1(2),TRACE AND STATUS(1),3;
NFG             SYNTAX 2(1),ARITHMETIC 1(1),2;
NFWLINE         OUTPUT 1(2),OUTPUT 2(2),READER 4(1),SEGMENTS(1),6;
NFWNAME         SYNTAX 1(2),SYNTAX 2(3),SEGMENTS(1),TABLES AND SPECIAL(1),7;
NFWSCALE        RESCALE(1),1;
NFWSP           STACK AND LABELS(6),6;
NFXT            STACK AND LABELS(4),4;
NFXTCHAR        NASTY CODE(1),OUTPUT 1(2),READER 1(3),6;
NFXTPARAM       STACK AND PROCS(1),SYNTAX 1(3),SEGMENTS(1),PROCEDURES AND ARRAYS(1),6;
NFXTPSET        SYNTAX 1(4),PROCEDURES AND ARRAYS(1),5;
NFXTPTYPE       PROCEDURE CALLS(1),SYNTAX 2(1),2;
NISACC          SYNTAX 2(1),CALLS AND CODE(1),2;
NISMOD          SYNTAX 2(2),CALLS AND CODE(1),3;
NO              ANALYSER(3),3;
NOAFTER         SYNTAX 1(2),DECLARATIONS(1),3;
NOASSY          READER 2(2),READER 4(2),4;
NOBITS          SYNTAX 1(3),SYNTAX 2(1),DECLARATIONS(1),5;
NODS            READER 1(1),READER 2(5),READER 4(1),READER 5(4),11;
NONAME          STACK AND PROCS(1),SYNTAX 1(3),SYNTAX 2(4),ARITHMETIC 1(1),CALLS AND CODE(3),12;
NORM            READER 4(1),READER 5(2),3;
NOSTG           SYNTAX 1(1),SYNTAX 2(1),TABLES AND SPECIAL(1),3;
NOTFST          ANALYSER(2),2;
NUMBER          GLOBAL(1),RESCALE(4),ARRAYS AND EXPRESSIONS(3),READER 2(2),READER 3(3),READER 4(2),READER 5(11),
                TRACE AND STATUS(5),SEGMENTS(3),DECLARATIONS(5),TABLES AND SPECIAL(8),PROCEDURES AND ARRAYS(7),
                ANSWER AND FOR(1),CALLS AND CODE(2),57;
NUMBERSCALE     GLOBAL(1),RESCALE(1),READER 2(1),READER 4(1),READER 5(7),ANSWER AND FOR(1),12;

O               READER 1(2),2;
OC              READER 1(1),READER 5(1),2;
OCTALFRAC       READER 2(2),READER 5(1),3;
OCTALNO         READER 2(3),3;
OCTLOOP         OUTPUT 1(2),SEGMENTS(1),3;
OFFSET          GLOBAL(1),ARRAYS AND EXPRESSIONS(1),PROCEDURES AND ARRAYS(2),4;
OFFSTACK        STACK AND LABELS(2),CHAINS AND BLOCKS(1),ARRAYS AND EXPRESSIONS(1),TRACE AND STATUS(1),PROCEDURES AND ARRAYS(1),
                ARITHMETIC 1(1),DO AND IF(1),CALLS AND CODE(2),10;
OFFSTEXPR       SYNTAX 2(1),ARITHMETIC 1(1),2;
OI              READER 1(1),READER 5(1),2;
OK              ANALYSER(2),READER 1(10),READER 4(8),READER 5(2),22;
OLDSCALE        RESCALE(1),1;
```



```
OLDSP           STACK AND LABELS(3),3;
ONERIT          SYNTAX 2(1),ARITHMETIC 1(1),2;
ONFDIM          SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
ONLAB           STACK AND LABELS(3),3;
ONSTACK         STACK AND LABELS(2),CHAINS AND BLOCKS(1),STACK AND PROCS(1),ARRAYS AND EXPRESSIONS(1),PROCEDURE CALLS(1),
                TRACE AND STATUS(1),SEGMENTS(1),ANSWER AND FOR(1),CALLS AND CODE(1),10;
OP              ARITHMETIC(6),6;
OPERATE         SUBSCRIPTS AND SCALING(1),ARITHMETIC 1(4),5;
OQ              READER 5(2),2;
ORACT           SYNTAX 1(1),TRACE AND STATUS(1),2;
ORF             SYNTAX 2(1),ARITHMETIC 1(1),2;
OUT             OUTPUT 1(3),OUTPUT 2(2),5;
OUT1014         OUTPUT 2(2),OUTPUT 3(1),3;
OUT1014CS       OUTPUT 2(1),TABLES AND SPECIAL(3),PROCEDURES AND ARRAYS(2),6;
OUT24           OUTPUT 2(3),STACK AND PROCS(1),4;
OUT24CS         OUTPUT 2(1),OUTPUT 3(2),ASSIGNMENT AND FOR(2),TABLES AND SPECIAL(2),PROCEDURES AND ARRAYS(1),
                DO AND IF(1),9;
OUT27           OUTPUT 3(2),CHAINS AND BLOCKS(1),ASSIGNMENT AND FOR(1),SEGMENTS(1),PROCEDURES AND ARRAYS(2),
                DO AND IF(1),CALLS AND CODE(1),9;
OUT5            OUTPUT 2(5),OUTPUT 3(2),STACK AND LABELS(2),CHAINS AND BLOCKS(2),ASSIGNMENT AND FOR(1),SEGMENTS(6),
                DECLARATIONS(1),DO AND IF(1),20;
OUTCHAR         OUTPUT 1(7),OUTPUT 2(1),READER 4(1),9;
OUTCJ           CHAINS AND BLOCKS(2),SYNTAX 1(1),TRACE AND STATUS(2),ANSWER AND FOR(1),CALLS AND CODE(1),7;
OUTCODE         SYNTAX 1(1),CALLS AND CODE(1),2;
OUTCONT         OUTPUT 2(1),OUTPUT 3(3),STACK AND LABELS(1),CHAINS AND BLOCKS(2),SEGMENTS(9),DO AND IF(1),17;
OUTNEV          GLOBAL(1),OUTPUT 1(1),OUTPUT 2(2),4;
OUTJ            OUTPUT 3(4),CHAINS AND BLOCKS(2),ARITHMETIC(1),ASSIGNMENT AND FOR(4),TRACE AND STATUS(1),PROCEDURES AND ARRAYS(3),
                ARITHMETIC 1(1),ARITHMETIC 2(1),ANSWER AND FOR(1),DO AND IF(3),21;
OUTPRESET       SYNTAX 1(1),TABLES AND SPECIAL(1),2;
OUTPRESETD      STACK AND PROCS(1),TABLES AND SPECIAL(2),3;
OUTUJ           CHAINS AND BLOCKS(2),PROCEDURES AND ARRAYS(1),DO AND IF(1),CALLS AND CODE(1),5;
OUTVCONST       OUTPUT 3(2),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),ASSIGNMENT AND FOR(1),DO AND IF(1),6;
OUTXFMN         OUTPUT 3(2),ACCUMULATORS(1),CHAINS AND BLOCKS(1),ARRAYS AND EXPRESSIONS(3),SUBSCRIPTS AND SCALING(2),
                ASSIGNMENT AND FOR(6),TRACE AND STATUS(1),PROCEDURES AND ARRAYS(1),ARITHMETIC 2(5),DO AND IF(2),
                CALLS AND CODE(1),27;
OVERBASE        GLOBAL(1),ARRAYS AND EXPRESSIONS(1),DECLARATIONS(2),PROCEDURES AND ARRAYS(2),6;
OVERLAY         GLOBAL(1),ARRAYS AND EXPRESSIONS(1),INITIALISATION(1),DECLARATIONS(4),PROCEDURES AND ARRAYS(1),8;
OVEROFF         SYNTAX 1(1),DECLARATIONS(1),2;
OVERON          SYNTAX 1(1),DECLARATIONS(1),2;
OVRTEST         SYNTAX 1(1),TRACE AND STATUS(1),2;
OX              READER 5(2),2;

P               DECLARATIONS(5),PROCEDURES AND ARRAYS(15),20;
PAC             STACK(1),STACK AND PROCS(6),PROCEDURE CALLS(2),SEGMENTS(1),PROCEDURES AND ARRAYS(1),CALLS AND CODE(2),12;
PARAMCLASS      PROCEDURE CALLS(3),3;
PARAMDECS       GLOBAL(1),PROCEDURES AND ARRAYS(2),3;
PARAMPTR        GLOBAL(1),STACK AND PROCS(5),6;
PARAMPTR2       GLOBAL(1),STACK AND PROCS(5),TABLES AND SPECIAL(1),7;
PARAMSPEC       STACK(1),STACK AND PROCS(3),PROCEDURE CALLS(4),CALLS AND CODE(3),11;
PARAMTAB        SYNTAX 1(2),TABLES AND SPECIAL(1),3;
PARTINT         SYNTAX 1(2),SYNTAX 2(1),DECLARATIONS(1),4;
PARTWORD        STACK(1),OUTPUT 3(1),STACK AND PROCS(2),ARRAYS AND EXPRESSIONS(5),SUBSCRIPTS AND SCALING(3),
                ARITHMETIC(3),ASSIGNMENT AND FOR(5),PROCEDURE CALLS(1),ARITHMETIC 1(4),ARITHMETIC 2(2),ANSWER AND FOR(7),
                DO AND IF(4),38;
PCN             STACK(1),STACK AND LABELS(9),TRACE AND STATUS(1),SEGMENTS(2),13;
PERM            ACCUMULATORS(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC(1),ARITHMETIC 2(4),DO AND IF(2),9;
PG              READER 1(1),READER 4(1),2;
PICK            ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(3),ARITHMETIC(3),ASSIGNMENT AND FOR(1),PROCEDURE CALLS(1),
                ARITHMETIC 1(1),ARITHMETIC 2(3),ANSWER AND FOR(2),CALLS AND CODE(2),19;
PLUSSUB         SYNTAX 1(2),SYNTAX 2(3),ARITHMETIC 1(1),6;
```

```
PNP             GLOBAL(1),PROCEDURE CALLS(9),CALLS AND CODE(7),17;
POWERTWO        ARITHMETIC(1),ARITHMETIC 2(8),9;
PR              READER 1(1),READER 4(1),2;
PREPACC         GLOBAL(1),ACCUMULATORS(2),SUBSCRIPTS AND SCALING(4),ARITHMETIC(2),PROCEDURE CALLS(2),ARITHMETIC 1(3),
                ARITHMETIC 2(1),ANSWER AND FOR(2),CALLS AND CODE(2),19;
PRESETOK        GLOBAL(1),STACK AND PROCS(1),INITIALISATION(1),DECLARATIONS(2),5;
PRESETSTRING    SYNTAX 1(1),TABLES AND SPECIAL(1),2;
PRINTADD        OUTPUT 1(1),OUTPUT 3(1),CHAINS AND BLOCKS(2),STACK AND PROCS(1),SEGMENTS(1),6;
PRINTBUFF       OUTPUT 1(2),OUTPUT 2(2),4;
PROC            OUTPUT 3(2),CHAINS AND BLOCKS(2),PROCEDURES AND ARRAYS(2),6;
PROCCHAIN       GLOBAL(1),STACK AND PROCS(1),PROCEDURES AND ARRAYS(1),3;
PROCEDURE       DEFINITIONS(1),PROCEDURE CALLS(3),PROCEDURES AND ARRAYS(1),CALLS AND CODE(1),6;
PROCENTRY       SYNTAX 1(2),PROCEDURES AND ARRAYS(1),3;
PROCPTR         GLOBAL(1),STACK AND PROCS(3),SEGMENTS(1),PROCEDURES AND ARRAYS(10),ANSWER AND FOR(2),17;
PROCSTACK       STACK AND PROCS(1),PROCEDURES AND ARRAYS(2),3;
PROCSTRING      GLOBAL(1),CHAINS AND BLOCKS(1),STACK AND PROCS(1),PROCEDURES AND ARRAYS(6),9;
PROCTRACE       SYNTAX 2(2),TRACE AND STATUS(1),3;
PSP             GLOBAL(1),PROCEDURE CALLS(21),CALLS AND CODE(4),26;
PTA             GLOBAL(1),OUTPUT 3(4),STACK AND LABELS(3),CHAINS AND BLOCKS(6),ASSIGNMENT AND FOR(1),ANSWER AND FOR(3),
                CALLS AND CODE(1),19;
PTF             STACK(1),1;
PTR             STACK AND LABELS(15),STACK AND PROCS(7),ARRAYS AND EXPRESSIONS(5),SEGMENTS(10),TABLES AND SPECIAL(10),47;
PUNCHLOOP       OUTPUT 2(3),3;
PVL             STACK(1),ARRAYS AND EXPRESSIONS(1),TABLES AND SPECIAL(1),ARITHMETIC 2(7),10;

QM              READER 1(1),READER 4(1),2;

R               NASTY CODE(4),READER 1(3),7;
RAISE           SYNTAX 2(1),ARITHMETIC 2(1),2;
READ            READER 1(2),READER 3(6),READER 4(1),READER 5(3),12;
READER          SYNTAX 1(1),READER 1(1),2;
READS           READER 1(4),4;
READT1          READER 1(1),READER 2(2),READER 4(1),READER 5(3),7;
READT2          READER 1(3),READER 2(5),READER 3(5),READER 4(8),READER 5(9),30;
READTAPE        NASTY CODE(1),READER 1(1),2;
RECNG           READER 1(1),READER 4(2),3;
RFF             OUTPUT 3(3),ACCUMULATORS(24),STACK AND LABELS(2),CHAINS AND BLOCKS(4),ARRAYS AND EXPRESSIONS(43),
                SUBSCRIPTS AND SCALING(14),ARITHMETIC(12),SEGMENTS(6),108;
RELADD          SYNTAX 2(1),CALLS AND CODE(1),2;
RELATION        SYNTAX 1(1),DO AND IF(1),2;
RFLOP           GLOBAL(1),READER 4(7),DO AND IF(7),15;
REPT            ANALYSER(7),READER 1(2),9;
RESCALE         RESCALE(2),ARRAYS AND EXPRESSIONS(1),ASSIGNMENT AND FOR(1),4;
REVCHAIN        CHAINS AND BLOCKS(1),TRACE AND STATUS(3),PROCEDURES AND ARRAYS(2),ANSWER AND FOR(1),CALLS AND CODE(1),8;
REVCJ           CHAINS AND BLOCKS(1),TRACE AND STATUS(3),4;
RH              GLOBAL(1),ACCUMULATORS(4),STACK AND PROCS(26),SUBSCRIPTS AND SCALING(38),ARITHMETIC(23),ASSIGNMENT AND FOR(16),
                PROCEDURE CALLS(26),SEGMENTS(3),DECLARATIONS(10),ARITHMETIC 1(25),ARITHMETIC 2(27),ANSWER AND FOR(34),
                DO AND IF(16),CALLS AND CODE(16),265;
RHACC           ACCUMULATORS(1),SUBSCRIPTS AND SCALING(4),ARITHMETIC(1),ARITHMETIC 2(1),DO AND IF(1),8;
RHSBITS         SYNTAX 2(1),ARITHMETIC 1(1),2;
RHSPTEST        PROCEDURE CALLS(1),SYNTAX 2(1),2;
RO              READER 1(6),READER 4(1),7;
RTA             GLOBAL(1),ASSIGNMENT AND FOR(4),ANSWER AND FOR(2),DO AND IF(2),9;
RTL             GLOBAL(1),ANSWER AND FOR(3),DO AND IF(3),7;

S               NASTY CODE(3),OUTPUT 1(4),CHAINS AND BLOCKS(2),SUBSCRIPTS AND SCALING(6),PROCEDURE CALLS(2),
                READER 1(9),READER 2(2),TRACE AND STATUS(2),ARITHMETIC 1(2),ARITHMETIC 2(4),DO AND IF(4),40;
S0              CHAINS AND BLOCKS(4),TRACE AND STATUS(2),DO AND IF(10),16;
S1              NASTY CODE(1),CHAINS AND BLOCKS(2),TRACE AND STATUS(2),DO AND IF(3),8;
S2              NASTY CODE(1),CHAINS AND BLOCKS(2),TRACE AND STATUS(2),DO AND IF(4),9;
```



```
S3              CHAINS AND BLOCKS(2),TRACE AND STATUS(2),DO AND IF(2),6;
S4              CHAINS AND BLOCKS(2),TRACE AND STATUS(2),DO AND IF(4),8;
S5              DO AND IF(5),5;
SC              READER 1(1),READER 3(1),2;
SCALE           ARRAYS AND EXPRESSIONS(8),ARITHMETIC 1(11),ARITHMETIC 2(26),45;
SCALECON        RESCALE(1),SYNTAX 1(1),SYNTAX 2(1),TABLES AND SPECIAL(2),5;
SCALEFIRM       GLOBAL(1),ARRAYS AND EXPRESSIONS(1),ARITHMETIC(1),PROCEDURE CALLS(3),ARITHMETIC 1(3),ARITHMETIC 2(1),
                ANSWER AND FOR(3),CALLS AND CODE(2),15;
SCALENUM        ARRAYS AND EXPRESSIONS(2),SUBSCRIPTS AND SCALING(1),ARITHMETIC(3),PROCEDURE CALLS(1),ANSWER AND FOR(1),8;
SCALETERM       SYNTAX 1(1),SYNTAX 2(3),ARITHMETIC 1(1),5;
SCALETEST       SUBSCRIPTS AND SCALING(1),ARITHMETIC 1(1),ARITHMETIC 2(2),4;
SCAN            PROCEDURES AND ARRAYS(3),3;
SCANLAB         STACK AND LABELS(2),SEGMENTS(1),3;
SELOPTA         ARITHMETIC(1),SYNTAX 2(1),2;
SEMICOM         READER 2(1),READER 3(3),READER 4(1),5;
SENSE           OUTPUT 3(2),2;
SFTANS          SYNTAX 2(1),ANSWER AND FOR(1),2;
SETASS          ARITHMETIC(1),SYNTAX 1(1),2;
SETCHAINTOPTA   STACK AND LABELS(3),CHAINS AND BLOCKS(1),PROCEDURES AND ARRAYS(1),DO AND IF(2),7;
SFTBUE          CHAINS AND BLOCKS(2),STACK AND PROCS(1),TRACE AND STATUS(1),SEGMENTS(1),PROCEDURES AND ARRAYS(1),
                ANSWER AND FOR(2),CALLS AND CODE(3),11;
SETLAB          SYNTAX 2(1),TRACE AND STATUS(1),2;
SFTLABEL        STACK AND LABELS(1),TRACE AND STATUS(1),SEGMENTS(1),PROCEDURES AND ARRAYS(1),4;
SFTLB           SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
SFTLEVEL        SYNTAX 2(1),TRACE AND STATUS(1),2;
SFTLHRH         STACK AND PROCS(3),ARRAYS AND EXPRESSIONS(1),PROCEDURE CALLS(1),ARITHMETIC 1(1),DO AND IF(1),
                CALLS AND CODE(1),8;
SFTLIBSEG       SYNTAX 1(3),PROCEDURES AND ARRAYS(1),4;
SFTLIBVAR       SYNTAX 1(1),SEGMENTS(1),2;
SFTNEZ          SYNTAX 2(1),DO AND IF(1),2;
SFTNO           SYNTAX 1(2),TRACE AND STATUS(1),3;
SFTPARAMS       SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
SFTPREF         SYNTAX 2(1),ARITHMETIC 1(1),2;
SFTRT           ASSIGNMENT AND FOR(1),ANSWER AND FOR(2),3;
SFTSHIFT        SYNTAX 2(1),ARITHMETIC 1(1),2;
SFTTEN          SYNTAX 1(1),SYNTAX 2(1),TRACE AND STATUS(1),3;
SFTTEST         SYNTAX 1(1),TRACE AND STATUS(1),2;
SFTUB           SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
SFTUPPROC       PROCEDURE CALLS(1),SYNTAX 2(1),2;
SFTUPSUB        SYNTAX 1(1),ARITHMETIC 1(1),2;
SFTYES          SYNTAX 1(6),SYNTAX 2(3),TRACE AND STATUS(1),10;
SGB             STACK(1),ARRAYS AND EXPRESSIONS(1),ARITHMETIC(3),ARITHMETIC 2(2),7;
SH              READER 1(6),READER 5(1),5;
SHIFT           ARRAYS AND EXPRESSIONS(12),ASSIGNMENT AND FOR(10),22;
SHIFTS          READER 2(2),ARITHMETIC 2(7),9;
SI              READER 1(5),5;
SIMPLEASS       SYNTAX 2(1),ANSWER AND FOR(1),2;
SIZE            STACK AND LABELS(6),4;
SKIP            STACK AND LABELS(2),2;
SKIPCHAIN       GLOBAL(1),CHAINS AND BLOCKS(5),ASSIGNMENT AND FOR(1),TRACE AND STATUS(2),ANSWER AND FOR(2),
                DO AND IF(2),CALLS AND CODE(1),14;
SKIPDTA         SYNTAX 1(1),DECLARATIONS(1),2;
SO              READER 1(1),READER 3(6),READER 4(2),9;
SPB             STACK(1),STACK AND PROCS(5),PROCEDURE CALLS(6),PROCEDURES AND ARRAYS(2),ANSWER AND FOR(1),15;
SPECCON         SYNTAX 1(2),TABLES AND SPECIAL(1),3;
SPECLAB         SYNTAX 1(2),TABLES AND SPECIAL(1),3;
SPECNUM         SYNTAX 1(3),TABLES AND SPECIAL(1),4;
SPECONE         SYNTAX 1(1),TABLES AND SPECIAL(1),2;
SPECREL         SYNTAX 1(1),TABLES AND SPECIAL(1),2;
SPECSTRING      OUTPUT 3(1),SYNTAX 1(1),DECLARATIONS(1),TABLES AND SPECIAL(1),PROCEDURES AND ARRAYS(1),5;
```

```
SPIEL          STACK(1),OUTPUT 3(12),STACK AND PROCS(3),ARRAYS AND EXPRESSIONS(1),ASSIGNMENT AND FOR(7),PROCEDURE CALLS(3),
               DECLARATIONS(3),ARITHMETIC 1(2),ANSWER AND FOR(5),DO AND IF(2),CALLS AND CODE(5),44;
SS             SYNTAX 1(1),ANALYSER(2),TRACE AND STATUS(5),DO AND IF(2),10;
SSEL           SYNTAX 1(1),ANALYSER(2),3;
SSPTR          SYNTAX 1(1),ANALYSER(5),8;
ST             READER 1(1),READER 3(1),2;
STA            GLOBAL(1),OUTPUT 2(2),OUTPUT 3(2),CHAINS AND BLOCKS(2),ASSIGNMENT AND FOR(2),TABLES AND SPECIAL(7),
               PROCEDURES AND ARRAYS(3),DO AND IF(1),20;
STACK          STACK(4),NASTY CODE(3),ACCUMULATORS(3),STACK AND LABELS(4),STACK AND PROCS(3),READER 1(1),READER 2(7),
               READER 3(7),ARITHMETIC 1(2),ARITHMETIC 2(2),CALLS AND CODE(1),37;
STACKCHECK     STACK AND LABELS(3),3;
STACKEXPR      ARRAYS AND EXPRESSIONS(1),SYNTAX 1(1),SYNTAX 2(1),ARITHMETIC 1(2),5;
STACKFINISH    DEFINITIONS(1),CHAINS AND BLOCKS(1),INITIALISATION(1),3;
STACKPOINTER   GLOBAL(1),STACK AND LABELS(7),STACK AND PROCS(3),SUBSCRIPTS AND SCALING(1),PROCEDURE CALLS(1),
               INITIALISATION(1),ARITHMETIC 1(3),CALLS AND CODE(1),18;
STACKSTART     DEFINITIONS(1),INITIALISATION(1),READER 2(1),3;
START          STACK AND LABELS(6),ARRAYS AND EXPRESSIONS(8),14;
STARTDEC       SYNTAX 1(2),DECLARATIONS(1),3;
STARTFOR       SYNTAX 2(1),ANSWER AND FOR(1),2;
STARTSEG       CHAINS AND BLOCKS(1),SEGMENTS(2),PROCEDURES AND ARRAYS(1),4;
STARTUP        OUTPUT 1(1),INITIALISATION(1),ANALYSER(1),3;
STATUS         GLOBAL(1),CHAINS AND BLOCKS(2),STACK AND PROCS(2),TRACE AND STATUS(13),SEGMENTS(2),
               PROCEDURES AND ARRAYS(6),ANSWER AND FOR(2),DO AND IF(4),32;
STATUSCHECK    CHAINS AND BLOCKS(1),SYNTAX 1(2),SYNTAX 2(5),TRACE AND STATUS(1),ANSWER AND FOR(1),DO AND IF(4),14;
STATUSCODE     CHAINS AND BLOCKS(1),SYNTAX 1(2),TRACE AND STATUS(1),4;
STATUSTEST     CHAINS AND BLOCKS(3),3;
STB            STACK(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC(3),ASSIGNMENT AND FOR(1),PROCEDURE CALLS(4),
               PROCEDURES AND ARRAYS(1),ARITHMETIC 1(2),ARITHMETIC 2(11),ANSWER AND FOR(3),27;
STEPUNT        SYNTAX 2(2),ANSWER AND FOR(1),3;
STOPOP         OUTPUT 2(1),REBCALE(1),STACK AND LABELS(1),CHAINS AND BLOCKS(1),STACK AND PROCS(2),PROCEDURE CALLS(3),
               SEGMENTS(2),DECLARATIONS(1),TABLES AND SPECIAL(2),PROCEDURES AND ARRAYS(2),ARITHMETIC 1(2),
               ANSWER AND FOR(4),CALLS AND CODE(3),25;
STORE          SYNTAX 1(1),ANSWER AND FOR(1),2;
STOREAWAY      ASSIGNMENT AND FOR(2),ANSWER AND FOR(2),4;
STORFZERO      SYNTAX 2(1),ANSWER AND FOR(1),2;
STRING         STACK(1),OUTPUT 1(5),OUTPUT 3(12),CHAINS AND BLOCKS(4),STACK AND PROCS(3),SEGMENTS(2),
               PROCEDURES AND ARRAYS(3),50;
STRINGEX       SYNTAX 2(3),DECLARATIONS(1),4;
SUB            SYNTAX 2(1),ARITHMETIC 2(1),DO AND IF(4),6;
SUBA           SYNTAX 2(1),ANSWER AND FOR(1),2;
SUBCOMMA       SYNTAX 2(1),ARITHMETIC 1(1),2;
SUBTERM        SUBSCRIPTS AND SCALING(1),ARITHMETIC 1(2),3;
SUBPIEL        ASSIGNMENT AND FOR(2),DO AND IF(1),3;
SUDP           NASTY CODE(1),ACCUMULATORS(1),CHAINS AND BLOCKS(1),STACK AND PROCS(2),SUBSCRIPTS AND SCALING(1),
               SEGMENTS(1),TABLES AND SPECIAL(2),ARITHMETIC 1(2),CALLS AND CODE(1),12;
SUDPOPT        SUBSCRIPTS AND SCALING(1),ARITHMETIC 1(3),ARITHMETIC 2(2),6;
SYNTAX         SYNTAX 1(1),ANALYSER(2),3;

T              GLOBAL(1),STACK AND LABELS(10),STACK AND PROCS(2),SUBSCRIPTS AND SCALING(4),PROCEDURE CALLS(1),
               READER 3(2),20;
T1             GLOBAL(1),ANALYSER(2),READER 1(2),READER 2(9),READER 3(7),READER 4(15),READER 5(8),ARITHMETIC 2(2),
               ANSWER AND FOR(1),CALLS AND CODE(2),49;
T2             GLOBAL(1),INITIALISATION(1),READER 1(8),READER 2(10),READER 3(31),READER 4(8),READER 5(10),69;
TA             OUTPUT 2(10),OUTPUT 3(2),12;
TABADD         SYNTAX 1(1),TABLES AND SPECIAL(1),2;
TABLED         GLOBAL(1),TABLES AND SPECIAL(2),3;
TABLEI         GLOBAL(1),TABLES AND SPECIAL(2),3;
TABLESIZE      SYNTAX 1(1),DECLARATIONS(1),2;
TAG            OUTPUT 1(2),OUTPUT 2(6),8;
TB             OUTPUT 2(5),OUTPUT 3(3),8;
```



```
TBN            OUTPUT 2(2),2;
TBST           GLOBAL(1),OUTPUT 2(5),ACCUMULATORS(3),INITIALISATION(1),TRACE AND STATUS(1),SEGMENTS(5),16;
TBSTSTRING     NASTY CODE(1),STACK AND LABELS(3),READER 2(1),5;
TBXT           OUTPUT 1(6),OUTPUT 2(6),OUTPUT 3(2),CHAINS AND BLOCKS(2),STACK AND PROCS(1),READER 4(1),SEGMENTS(2),20;
TBXTLINE       OUTPUT 1(4),OUTPUT 2(2),OUTPUT 3(1),CHAINS AND BLOCKS(3),STACK AND PROCS(1),ANALYSER(1),12;
TFLAG          ASSIGNMENT AND FOR(4),4;
TGD            STACK(1),SUBSCRIPTS AND SCALING(1),TABLES AND SPECIAL(1),3;
TGI            STACK(1),1;
THEAD          OUTPUT 2(3),CHAINS AND BLOCKS(2),SEGMENTS(2),7;
THENEX         SYNTAX 2(1),CALLS AND CODE(1),2;
TIMES          OUTPUT 2(1),1;
TO             NASTY CODE(1),READER 2(2),3;
TOPACC         ACCUMULATORS(1),PROCEDURE CALLS(1),CALLS AND CODE(2),4;
TOPTEN         GLOBAL(1),OUTPUT 2(1),TRACE AND STATUS(1),TABLES AND SPECIAL(2),PROCEDURES AND ARRAYS(1),6;
TOSADD         STACK AND PROCS(1),DECLARATIONS(1),TABLES AND SPECIAL(1),3;
TRACE          GLOBAL(1),STACK AND LABELS(1),ARITHMETIC(1),INITIALISATION(1),TRACE AND STATUS(4),PROCEDURES AND ARRAYS(2),
               ANSWER AND FOR(1),DO AND IF(1),12;
TRACESTRING    OUTPUT 3(1),STACK AND LABELS(1),CHAINS AND BLOCKS(1),3;
TSUM           GLOBAL(1),OUTPUT 2(3),CHAINS AND BLOCKS(1),SEGMENTS(1),6;
TTAIL          OUTPUT 2(2),CHAINS AND BLOCKS(1),SEGMENTS(1),4;
TVP            STACK(1),ARRAYS AND EXPRESSIONS(3),SUBSCRIPTS AND SCALING(3),ARITHMETIC(4),ASSIGNMENT AND FOR(1),
               PROCEDURE CALLS(3),SEGMENTS(3),DECLARATIONS(1),TABLES AND SPECIAL(1),PROCEDURES AND ARRAYS(2),
               ARITHMETIC 1(2),ARITHMETIC 2(2),ANSWER AND FOR(4),CALLS AND CODE(1),33;
TYPE           CHAINS AND BLOCKS(6),PROCEDURE CALLS(3),9;
TYPEARRAY      SYNTAX 1(4),SYNTAX 2(3),DECLARATIONS(1),8;
TYPEBITS       STACK(1),ACCUMULATORS(3),STACK AND PROCS(7),ARRAYS AND EXPRESSIONS(6),SUBSCRIPTS AND SCALING(6),
               ARITHMETIC(9),ASSIGNMENT AND FOR(8),PROCEDURE CALLS(11),SEGMENTS(2),DECLARATIONS(3),TABLES AND SPECIAL(1),
               ARITHMETIC 1(3),ARITHMETIC 2(22),ANSWER AND FOR(7),DO AND IF(1),CALLS AND CODE(7),97;
TYPEFLOAT      SYNTAX 1(1),DECLARATIONS(1),2;
TYPEIARRAY     SYNTAX 1(4),DECLARATIONS(1),5;
TYPEINT        SYNTAX 1(7),SYNTAX 2(2),DECLARATIONS(1),10;
TYPEISWITCH    SYNTAX 1(3),DECLARATIONS(1),4;
TYPELAB        SYNTAX 1(3),SYNTAX 2(2),DECLARATIONS(1),6;
TYPELOC        SYNTAX 1(2),DECLARATIONS(1),3;
TYPEPROC       SYNTAX 1(6),SYNTAX 2(3),DECLARATIONS(1),10;
TYPESPEC       SYNTAX 1(4),DECLARATIONS(1),5;
TYPESWITCH     SYNTAX 1(1),SYNTAX 2(2),DECLARATIONS(1),4;
TYPETPROC      SYNTAX 1(5),SYNTAX 2(3),DECLARATIONS(1),9;
TYPEVPROC      SYNTAX 1(3),DECLARATIONS(1),4;
TYPEXPR        SYNTAX 2(1),ARITHMETIC 1(1),2;

UJBACK         ASSIGNMENT AND FOR(2),DO AND IF(2),4;
ULW            READER 1(5),5;
ULWORD         READER 1(1),READER 4(1),2;
ULWORDS        READER 1(2),2;
UNARYMINUS     SUBSCRIPTS AND SCALING(1),ARITHMETIC(1),SYNTAX 1(1),ANSWER AND FOR(2),DO AND IF(2),7;
UNS            STACK(1),1;
UNSFIELD       SYNTAX 1(1),SYNTAX 2(1),TABLES AND SPECIAL(1),3;
USELAB         STACK AND LABELS(1),ARRAYS AND EXPRESSIONS(1),TRACE AND STATUS(2),4;

V              SUBSCRIPTS AND SCALING(4),4;
VARCHECK       SYNTAX 1(1),SYNTAX 2(2),ANSWER AND FOR(1),4;

WARN           OUTPUT 2(1),ARRAYS AND EXPRESSIONS(1),SUBSCRIPTS AND SCALING(1),ARITHMETIC 2(1),4;
WDS            STACK(1),SEGMENTS(1),2;
WFAR           ANSWER AND FOR(2),2;
WHILEL         SYNTAX 2(1),ANSWER AND FOR(1),2;
WORD           OUTPUT 1(6),OUTPUT 2(8),14;
```

```
X               NASTY CODE(1),OUTPUT 3(2),READER 4(2),5;
XFM             OUTPUT 3(2),2;
XSPARAMS        STACK AND PROCS(2),SEGMENTS(1),3;
XX              ASSIGNMENT AND FOR(1),1;

Y               NASTY CODE(1),1;
YES             ANALYSER(2),2;

Z               NASTY CODE(1),1;
ZERO            READER 5(3),3;
ZEROACCS        ACCUMULATORS(1),STACK AND LABELS(2),CHAINS AND BLOCKS(2),ARITHMETIC(2),ASSIGNMENT AND FOR(2),
                PROCEDURES AND ARRAYS(1),ARITHMETIC 1(1),ANSWER AND FOR(2),DO AND IF(1),CALLS AND CODE(1),15;
ZEROARRAYS      SYNTAX 1(1),PROCEDURES AND ARRAYS(1),2;
ZPROCOMP        SYNTAX 2(1),DO AND IF(1),2;
ZPRONUM         SYNTAX 1(2),TRACE AND STATUS(1),3;
```



```
'BOOLWIDTH' 3
'START'
'TERMINAL' 'SYMBOL' ZERO,INT,INTCON,REALCON,PCONST,SHIFT,RO,STRING,INST,DIFFER,END,IF,COLON,SEMI,THEN,OVERFLOW,NO,TRAC
ASSIGNMENT,LOOP,AND,OR,UNION,LOCATION,MASK,SPECIAL,ORB,ORBOSB,CRB,CRBCSB,MULT,PLUS,COMMA,MINUS,SELF,DIV,ANYACC,MODACC,
RAY,BEGIN,CODE,DO,ELSE,FOR,GOTO,COMMON,INTEGER,FIXED,FLOATING,LABEL,PROCEDURE,ID,UNSIGNED,PRESET,SWITCH,LIBRARY,STEP,T
LE,UNTIL,VALUE,WHILE,EXTERNAL,ABSOLUTE,OVERLAY,OSB,ANSWER,CSB,POWER,FINISH,BECOMES,WITH,COMPILE,BITS,COMMAB,LOAD,DUMP,
COVER,LEVEL,TEST;
'CLASS' 'NAME' DIFFERENCE,024,060A,F1,F2,F3,A1,A2,A3,A4,A5,06A,AXPRA,COMSPECID,COMSPECIDLIST,COMSPECIDCONT,NPART,NREAD
WRITE,NSHIFT,NJUMP,NLOC,NW1,NW2,NW3,RUN,COMPILEITEM,COMMONDEC,COMMONLIST,COMMONITEM,TYPE,059,SGINT,022,021,IDLIST,NEWI
052,ARRAYLIST,BOUNDPAIR,MOREBOUNDS,ARRAYTAIL,042,PRESETLIST,PRESETS,PRESETNUM,PNUMBER,040,COMPROCS,PARAMLEV2,057,PARAM
PELIST,PARAMTYPE,054,053,055,056,020,VALPROC,TABLEDEC,TABLEDETAIL,FIELDLIST,FIELDTYPE,PARTFIELD,INTFIXFIELD,06U,OVERLA
EC,BASE,043,DATADEC,SPECIALDEC,SPECLIST,SPECITEM,ZINT,045,SPECADD,044,MORESPEC,046,023,PROGRAM,BODY,DL,D,047,PROCREST,
RAMPTERPART,PARAMETERS,PARAMETERSET,063,PROCPARAMLIST,058,062,PAIRLIST,061,064,065,PROCBODY,ODL,CSL,CS,COMPOUND,037,SL
T,S,IFSTAT,CONDITIONLIST,CONDITION,YESNO,LCOND,AXPR,TERM,LOCVAR,029,SUBSCRIPT,SUBSTART,SUBEX,SUBTERMS,SUBEXCONT,CEXPR,
PR,MORESUBS,SINGLESUBS,032,FACTOR,UNITED,COLLATION,TERT,SEC,BITSET,ONEORMORE,PRI,TPRI,024,RHSPSEL,PROCCALL,01,SHIFTS,0
03,04,05,06,RCOND,030,CONDTERM,CONDTERMS,031,FORSTAT,ANSWERSTAT,GOTOSTAT,027,035,038,033,025,026,LHSB,039,034,051,SS,L
ELDEC,TRACETYPE,SWLIST,041,048,049,050,069,LIBLIST,LIBITEM,018,LIBIDLIST,LIBID,016,LIBPROCS,017,019,066,067,EXTLIST,EX
TEM,014,09,EXTIDLIST,EXTID,012,EXTPROCS,013,015,ABSLIST,ABSITEM,010,ABSIDLIST,ABSID,07,ABSPROCS,08,011,068,070,SELPARA
,PARAMLOOP,SELPTYPE,SELMORE,MULTITEST;
```



```
'SELECTOR' 'ACTION'

ANYMORE,
INSTTYPE,
NEXTPTYPE,
NMSPTEST,
SPLOPTA,
SPTASS,
SPTUPPROC,


BEGINBLOCK,
SPHEAD,
ENDBLOCK,
ENTERPROC,
FINISHSEG,
KILLEXPR,
LOCACT,
LOOKUP,
NPXTPARAM,
NONAME,
OUTCJ,
SCALECON,
SPECSTRING,
STACKEXPR,
STATUSCHECK,
STATUSCODE,
UNARYMINUS;
```

'ACTION'

ARSADD,
AND,
ANDA,
ANDBSPEC,
ANDBSU,
ANSCHECK,
ANYPREF,
ARSCV,
ASSTRACE,
AVMCHECK,
BFGINPROC,
BFGINPROG,
BFGINPSPEC,
BITSIN,
CALLPROC,
CHECKCV,
CLEARTYPE,
CNDFLAB,
CODPSHIFT,
COMOFF,
COMON,
CONDADD,
CONDMINUS,
CONDPLUS,
CONDSUB,
CONSTANT,
DFCSIZE,
DIVIDE,
DOCS,
DOSHIFT,
ELSES,
ELSEX,
ELSFFI,
ENDARRAY,
ENDBECS,
ENDFOR,
ENDPROC,
ENDPROG,
ENDPSPEC,
ENDST,
EXITCHECK,
EXPRTYPE,
EXTADD,
FI,
FIELDDISP,
FIELDPOSN,
FIEX,
FINISHPROC?
FIRSTDIM,
FORTRACE,
GOTOL,
GOTOSK,
IFEX,
IFSI

'ACTION'

INCYOS,
KILLPARAMS,
LABL,
LABSK,
LABTRACE,
LASTDIM,
LHSBITS,
LHSPROC,
LIBADD,
LOOKUPD,
MIDDIM,
MINUSSUB,
MOREFOR,
MPY,
MSK,
MULTICALL,
NPENUM,
NPQ,
NPWNAME,
NPXTPSET,
NISACC,
NISMOD,
NOAFTER,
NOBITS,
NOBIG,
OFFSTEXPR,
ONBBIT,
OVERIM,
OVACT,
OWP,
OUTCODE,
OUTPRESET,
OVEROFF,
OVERON,
OVRTEST,
PARAMTAB,
PARTINT,
PLUSSUB,
PRESETSTRING,
PROCENTRY,
PROCTRACE,
RAISE,
RELADD,
RPLATION,
RWBITS,
SCALETERM,
SETANS,
SFTLAB,
SFTLO,
SFTLEVEL,
SFTLIBSEG,
SFTLIBVAR;

```
'ACTION'

SETNEZ,
SFTNO,
SETPARAMS,
SFTPREF,
SETSHIFT,
SETTEN,
SFTTEST,
SETUB,
SETUPSUB,
SFTYES,
SIMPLEASS,
SKIPDTA,
SPECCON,
SPECLAB,
SPECNUM,
SPECONE,
SPECREL,
STARTDEC,
STARTFOR,
STEPUNT,
STORE,
STOREZERO,
STRINGEX,
SUB,
SUBA,
SUBCOMMA,
TABADD,
TABLESIZE,
THENEX,
TYPEARRAY,
TYPEFLOAT,
TYPEIARRAY,
TYPEINT,
TYPEISWITCH,
TYPELAB,
TYPELOC,
TYPEPROC,
TYPESPEC,
TYPESWITCH,
TYPETPROC,
TYPEVPROC,
TYPEXPR,
UNSFIELD,
VARCHECK,
WHILEL,
ZPROARRAYS,
ZEROCOMP,
ZPRONUM;
```



```
'SYMBOL' 'DEFINITION'
   ZERO=(0)
   INT=(0,1)
   INTCON=(1)
   REALCON=(2)
   PCONST=(0,1,2)
   SHIFT=(3)
   BO=(4)
   STRING=(5)
   INST=(6)
   DIFFER=(7)
   END=(8)
   IF=(9)
   COLON=(10)
   SEMI=(11)
   THEN=(12)
   OVERFLOW=(13)
   DO=(14)
   TRACE=(15)
   ASSIGNMENT=(16)
   LOOP=(17)
   AND=(18)
   OR=(19)
   UNION=(20)
   LOCATION=(21)
   MASK=(22)
   SPECIAL=(23)
   ABR=(24)
   ABRPOSN=(24,59)
   PBR=(25)
   PBRPSN=(25,01)
   MULT=(26)
   PLUS=(27)
   COMMA=(28)
   MINUS=(29)
   SELF=(30,26)
   DIV=(31)
   ANYACC=(32)
   CODACC=(32)
```

```
ARRAY=(33)
BEGIN=(34)
CODE=(35)
DO=(36)
ELSE=(37)
FOR=(38)
GOTO=(39)
COMMON=(40)
INTEGER=(41)
FIXED=(42)
FLOATING=(43)
LABEL=(44)
PROCEDURE=(45)
ID=(46)
UNSIGNED=(47)
PRESET=(48)
FINISH=(49)
SWITCH=(49)
LIBRARY=(50)
STEP=(51,10)
TABLE=(52)
UNTIL=(53,10)
VALUE=(54)
WHILE=(55)
EXTERNAL=(56)
ABSOLUTE=(57)
OVERLAY=(58)
OSB=(59)
ANSWER=(60)
CSB=(61)
POWER=(62)
RECOMES=(63)
WITH=(64)
COMPILE=(65)
BITS=(66)
COMMAB=(66,28)
LOAD=(67)
DUMP=(68)
RECOVER=(69)
LEVEL=(70)
TEST=(71)
```



```
'SYNTAX' 'RULE'

RUN=(COMPILEITEM,070)

070=(FINISH)(SEMI,RUN)

COMPILEITEM=()
   (COMMONDEC)
   (PROGRAM)
   (LEVELDEC)
   (YESNO,TRACETYPE,TRACE)
   (LIBRARY,064)
   (EXTERNAL,ORR,EXTLIST,CRB)
   (ABSOLUTE,ORR,ABSLIST,CRB)
   (TEST,SETTEST,068)

COMMONDEC=(COMMON,ORB,COMON,COMMONLIST,COMOFF,CRB)

COMMONLIST=(STARTDEC,COMMONITEM,023)

COMMONITEM=()
   (TYPE,022)
   (VALPROC,COMPROCS)
   (TABLEDEC,PRESETLIST)
   (OVERLAYDEC)
   (SPECIALDEC)
   (LABEL,TYPELAB,COMSPECIDLIST)
   (SWITCH,TYPEISWITCH,COMSPECIDLIST)
   (SETYES,TYPEPROC,PROCEDURE,COMPROCS)

TYPE=(INTEGER,059)
   (FIXED,ORBOSB,INT,NOBITS,COMMA,SGINT,NOAFTER,CRBCSB)
   (FLOATING,TYPEFLOAT)

059=(OSB,INT,NOBITS,CSB,PARTINT)
   (TYPEINT)

SGINT=(INT)
   (PLUS,INT)
   (MINUS,INT,NEGNUM)

022=(021,PRESETLIST)
   (TYPETPROC,PROCEDURE,COMPROCS)

021=(IDLIST,DECSIZE)
   (ARRAY,TYPEARRAY,ARRAYLIST)

IDLIST=(NEWID,052)

NEWID=(ID,NEWNAME)

052=()
   (COMMA,IDLIST)

ARRAYLIST=(ZEROARR,YS,IDLIST,OSB,BOUNDPAIR,MOREBOUNDS,CSB,ENDARRAY,042)

COMSPECIDLIST=(COMSPECID,COMSPECIDCONT)

COMSPECIDCONT=()
   (COMMA,COMSPECIDLIST)
```

```
BOUNDPAIR=(SGINT,SETLB,COLON,SGINT,SETUB)

VARBOUNDS=(ONEDIM)
   (FIRSTDIM,COMMA,BOUNDPAIR,ARRAYTAIL)

ARRAYTAIL=(MIDDIM,COMMA,BOUNDPAIR,ARRAYTAIL)
   (LASTDIM)

062=()
   (COMMA,ARRAYLIST)

PRESETLIST=()
   (BECOMES,SKIPDTA,PRESETS)

PRESETS=(PRESETNUM,040)

PRESETNUM=(STRING,PRESETSTRING)
   (PNUMBER,OUTPRESET)
   (ORR,PRESETS,CRB)

PNUMBER=(PCONST)
   (PLUS,PCONST)
   (MINUS,PCONST,NEGNUM)
   (ZERONUM)

040=()
   (COMMA,PRESETS)

COMPROCS=(COMSPECID,PARAMLEV2,020)

PARAMLEV2=(BEGINPSPEC,057)

057=(ORB,PARAMTYPELIST,CRB,ENDPSPEC)
   (ENDPSPEC)

PARAMTYPELIST=(PARAMTYPE,NEXTPARAM,056)

PARAMTYPE=(TYPE,054)
   (LOCATION,TYPELOC,053)
   (LABEL,TYPELAB)
   (SWITCH,TYPESWITCH)
   (TABLE,TYPEINT,TYPEARRAY)
   (VALUE,055)
   (SETYES,TYPEPROC,PROCEDURE)

054=(ARRAY,TYPEARRAY)
   (TYPETPROC,PROCEDURE)

053=(TYPE)
   (TYPESPEC,NEXTPARAM)

055=(TYPE)
   (TYPESPEC,NEXTPARAM)
   (TYPEVPROC,PROCEDURE)

COMSPECID=(ADDRSPEC,NEWID,SPECONE)
```



```
056=()
   (COMMA,NEXTPSET,PARAMTYPELIST)

020=()
   (COMMA,COMPROCS)

VALPROC=(VALUE,TYPEVPROC,PROCEDURE)

TABLEDEC=(TABLE,TYPEINT,TYPEARRAY,NEWID,OSB,INT,TABLESIZE,CSB,TABLEDETAIL,CLEARTYPE,TYPEINT)

TABLEDETAIL=(OSB,TABADD,FIELDLIST)

FIELDLIST=(CLEARTYPE,ID,FIELDTYPE,TYPEARRAY,NEWNAME,060)

FIELDTYPE=(TYPE,SGINT,FIELDDISP)
   (PARTFIELD)
   (UNSIGNED,PARTFIELD,UNSFIELD)

PARTFIELD=(ORBOSB,INT,NOBITS,NOSIG,INTFIXFIELD,CRBCSB,SGINT,FIELDDISP,COMMAB,INT,FIELDPOSN)

INTFIXFIELD=(COMMA,SGINT,NOAFTER)
   (PARTINT)

060=(CSR)
   (SEMI,060A)

060A=(060)
   (FIELDLIST)

OVERLAYDEC=(OVERLAY,BASE,WITH;OVERON,DATADEC,OVEROFF)

BASE=(ID,LOOKUPD,063)
   (NONAME,OSB,INT,INCTOS,CSB)

063=()
   (OSB,SGINT,INCTOS,CSB)

DATADEC=(TYPE,021)
   (TABLEDEC)

SPECIALDEC=(SPECIAL,ARRAY,TYPEINT,TYPEARRAY,ADDRSPEC,NEWID,BECOMES,SPECLIST)

SPECLIST=(SPECITEM,066)

SPECITEM=(ZINT,065)
   (STRING,SPECSTRING)
   (CLEARTYPE,TYPE,ORB,PNUMBER,SPECNUM,MORESPEC)

ZINT=(SGINT)
   (ZERONUM)

065=(SETTEM,SPECADD)
   (CLEARTYPE,TYPEINT,SPECNUM)

SPECADD=(COLON,ID,SPECLAB)
   (SELF,SGINT,SPECREL)
   (DIV,066)
```

```
044=(SCALAR,NAME,FACTOR,SPECCON)
   (BASE,SPECCON)

MORESPEC=(CRB)
   (COMMA,NUMBER,SPECNUM,MORESPEC)

046=()
   (COMMA,SPECLIST)

023=()
   (SEMI,COMMONLIST)

PROGRAM=(ID,BEGINPROG,BEGIN,BODY,ENDPROG,END)

BODY=(DL,ENDDECS,SL)

DL=(STARTDEC,D,SEMI,DSC)

D=(TYPE,047)
   (VALPROC,NEWID,BEGINPROC,COLON,NEWID,NEXTPSET,PROCREST)
   (TABLEDEC,PRESETLIST)
   (OVERLAYDEC)
   (SPECIALDEC)
   (LEVELDEC)
   (NO,SETNO,TRACETYPE,TRACE)
   (SWITCH,ADDRSW,TYPESWITCH,NEWID,BECOMES,SWLIST)
   (SETYES,TYPEPROC,048)

047=(021,PRESETLIST)
   (TYPETPROC,PROCEDURE,NEWID,BEGINPROC,PROCREST)

PROCREST=(PARAMETERPART,SEMI,PROCBODY,ENDPROC)

PARAMETERPART=()
   (ORB,PARAMETERS,CRB)

PARAMETERS=(PARAMETERSET,065)

PARAMETERSET=(TYPE,063)
   (LOCATION,TYPELOC,062)
   (LABEL,TYPELAB,IDLIST)
   (SWITCH,TYPEISWITCH,IDLIST)
   (TABLE,TYPEINT,TYPEIARRAY,NEWID,OSB,INT,CSB,PARAMTAB,TABLEDETAIL,PARAMTAB)
   (VALUE,064)
   (SETYES,TYPEPROC,PROCEDURE,PROCPARAMLIST)

063=(ARRAY,TYPEIARRAY,IDLIST)
   (TYPETPROC,PROCEDURE,PROCPARAMLIST)

PROCPARAMLIST=(NEWID,PARAMLEV2,058)

058=()
   (COMMA,PROCPARAMLIST)

062=(TYPE,IDLIST)
   (TYPESPEC,PAIRLIST)
```



```
PAIRLIST=(NEWID,COLON,NEWID,061)

061=()
   (COMMA,PAIRLIST)

064=(TYPE,IDLIST)
   (TYPESPEC,PAIRLIST)
   (TYPEVPROC,PROCEDURE,PROCPARAMLIST)

065=()
   (SEMI,NEXTPSET,PARAMETERS)

PROCBODY=(CODE,BEGIN,KILLPARAMS,ODL,ENTERPROC,CSL,END)
   (SETPARAMS,051)

ODL=(DL,ENDDECS)
   (ENDDECS)

CSL=(CS,034)

CS=()
   (COMPOUND,STATUSCODE)
   (INST,INSTTYPE,NPART,OUTCODE)
   (ID,CODELAB,COLON,CS)

COMPOUND=(STATUSCHECK,BEGIN,037)

037=(SL,END)
   (BEGINBLOCK,BODY,END,ENDBLOCK)

SL=(ST,039)

039=(S,ENDST)

ST=()
   (IFSTAT)
   (FORSTAT)
   (ASSIGNSTAT)
   (GOTOSTAT)
   (COMPOUND)
   (CODE,STATUSCODE,BEGIN,035)
   (ID,038)
   (STATUSCHECK,026,A1)

IFSTAT=(IF,IFS,CONDITIONLIST,THEN,ENDST,ST,ELSE,ELSES,S,FI)

CONDITIONLIST=(CONDITION,031)

CONDITION=(VOSSO,UVPREFLOW,UVRTEST)
   (STACKPROP,ANYPREP,LCOND,RCOND,RELATION)

VOSSO=(L,STEW)
   (IF VFS,TYPEPROC)
```

```
LCOND=(AXPR)
   (ZERO,CONSTANT)

AXPR=(TERM,06)
   (AXPRA)

AXPRA=(PLUS,TERM,06)
   (MINUS,TERM,UNARYMINUS,06)

TERM=(FACTOR,05,SCALETERM)

LOCVAR=(ID,LOOKUP,029)
   (NONAME,SINGLESUBS)

029=(SUBSCRIPT)
   (AYMCHECK)

SUBSCRIPT=(SUBSTART,MORESUBS,CSB,KILLEXPR)

SUBSTART=(OSB,SETUPSUB,SUBEX)

SUBEX=(SUBTERMS)
   (CEXPR,PLUSSUB)
   (TERM,PLUSSUB,SUBEXCONT)
   (ZERO)

SUBTERMS=(PLUS,TERM,PLUSSUB,SUBEXCONT)
   (MINUS,TERM,MINUSSUB,SUBEXCONT)

SUBEXCONT=()
   (SUBTERMS)

CEXPR=(IF,IFEX,CONDITIONLIST,THEN,THENEX,EXPR,ELSE,ELSEFI,ELSEX,EXPR,ELSEF1,FIEX)

EXPR=(CEXPR)
   (AXPR)
   (ZERO,CONSTANT,SCALETERM)
   (STRING,STRINGEX,SCALETERM)

MORESUBS=()
   (COMMA,SUBCOMMA,SUBEX,MORESUBS)

SINGLESUBS=(SUBSTART,CSB,KILLEXPR)

032=(LOOKUP,SINGLESUBS,LABSK)
   (LABL)

FACTOR=(DIFFERENCE,024)

024=()
   (POWER,DIFFERENCE,RAISE,024)

DIFFERENCE=(UNITED,04)
UNITED=(COLLATION,03)
```



```
COLLATION=(TERT,02)

TERT=(SEC,01)
   (PRI,01)

SEC=(BITSET,BITSIN,PRI,RHSBITS)

BITSET=(BITS,CLEARTYPE,OSB,INT,ONEORMORE,FIELDPOSN,UNSFIELD,CSB)

ONEORMORE=(NOBITS,NOSIG,PARTINT,COMMA,INT)
   (ONEBIT)

PRI=(TPRI)
   (REALCON,CONSTANT)
   (ORB,STACKEXPR,SETPREF,EXPR,CRB,OFFSTEXPR)

TPRI=(INTCON,CONSTANT)
   (ID,LOOKUP,028)
   (CLEARTYPE,TYPE,TYPEXPR,ORB,EXPR,CRB,EXPRTYPE)
   (NONAME,SINGLESUBS)
   (LOCATION,ORB,LOCVAR,LOCACT,CRB)
   (LABEL,ORB,ID,032,CRB)

028=(SUBSCRIPT,VARCHECK)
   (RHSPTEST,RHSPSEL)

01=()
   (SHIFT,SETSHIFT,SHIFTS,DOSHIFT,01)

SHIFTS=(SEC,PLUSSUB)
   (TPRI,SCALETERM,PLUSSUB)
   (ORB,SUBEX,CRB)

02=()
   (MASK,TERT,MSK,02)

03=()
   (UNION,COLLATION,ORF,03)

04=()
   (DIFFER,UNITED,NEG,04)

05=()
   (MULT,FACTOR,MPY,05)
   (DIV,FACTOR,DIVIDE,05)

0A=()
   (06A)

06A=(PLUS,TERM,ADD,06)
   (MINUS,TERM,SUB,06)

RCOND=(RO,030)
   (SETNEZ)

030=(CONDTERM,CONDTERMS)
   (ZERO,ZEROCOMP)

CONDTERMS=(TERM,CONDPLUS)
   (PLUS,TERM,CONDPLUS)
   (MINUS,TERM,CONDMINUS)
```

```
CONDTERMS=()
   (PLUS,TERM,CONDADD,CONDTERMS)
   (MINUS,TERM,CONDSUB,CONDTERMS)

0  =()
   ,ND,OUTCJ,CONDITIONLIST)
   (OR,CRACT,CONDITIONLIST)

ANSWERSTAT=(STATUSCHECK,ANSWER,SETANS,EXPR,ANSCHECK)

GOTOSTAT=(GOTO,ID,027)

027=(LOOKUP,STATUSCHECK,SINGLESUBS,LABSK,GOTOSK)
   (GOTOL)

035=(CSL,END)
   (BEGIN@LOCK,DL,ENDDECS,CSL,END,ENDBLOCK)

038=(LOOKUP,STATUSCHECK,033)
   (SETLAB,COLON,S)

033=(025,A1)
   (LHSPROC,PROCCALL,BEHEAD)

025=()
   (SUBSCRIPT)

026=(BITSET,LHSB)
   (NONAME,SINGLESUBS)

LHSB=(ID,LOOKUP,LHSBITS,025)
   (NONAME,LHSBITS,SINGLESUBS)

039=()
   (SEMI,SL)

034=()
   (SEMI,CSL)

051=(BEGIN;ODL,PROCENTRY,SL,END,EXITCHECK)
   (ENDDECS,PROCENTRY,SS,EXITCHECK)

99=()
   (IFSTAT)
   (FORSTAT)
   (ANSWERSTAT)
   (GOTOSTAT)
   (ID,LOOKUP,STATUSCHECK,033)
   (STATUSCHECK,026,A1)
```



```
LEVELDEC=(LEVEL,INT,SETLEVEL)

TRACETYPE=(ASSIGNMENT,ASSTRACE)
   (LOOP,FORTRACE)
   (LABEL,LABTRACE)
   (PROCEDURE,PROCTRACE)

SWLIST=(ID;SPECLAB,041)

041=()
   (COMMA,SWLIST)

048=(ASSIGNMENT,ASSTRACE;TRACE)
   (LOOP,FORTRACE,TRACE)
   (LABEL,LABTRACE,TRACE)
   (PROCEDURE,049)

049=(NEWID,BEGINPROC,PROCREST)
   (PROCTRACE,TRACE)

050=()
   (BL)

069=(ORB,LIBLIST,CRB)
   (CLEARTYPE,066)

LIBLIST=(CLEARTYPE,LIBITEM,019)

LIBITEM=()
   (TYPE,018)
   (VALPROC,LIBPROCS)
   (SETYES,TYPEPROC,PROCEDURE,LIBPROCS)

018=(LIBIDLIST)
   (TYPETPROC,PROCEDURE,LIBPROCS)

LIBIDLIST=(LIBID,016)

LIBID=(ID,DIV,INT,LIBADD,NEWNAME)

016=()
   (COMMA,LIBIDLIST)

LIBPROCS=(LIBID,PARAMLEV2,017)

017=()
   (COMMA,LIBPROCS)

019=()
   (SEMI,LIBLIST)
```

```
066=(TYPE,067)
   (VALPROC,LIBID,SETLIBSEG,COLON,NEWID,NEXTPSET,PROCREST,FINISHSEG)
   (SETYES,TYPEPROC,PROCEDURE,LIBID,SETLIBSEG,PROCREST,FINISHSEG)

067=(LIBID,BECOMES,PNUMBER,SCALECON,SETLIBVAR)
   (TYPETPROC,PROCEDURE,LIBID,SETLIBSEG,PROCREST,FINISHSEG,

EXTLIST=(CLEARTYPE,EXTITEM,015)

EXTITEM=()
   (TYPE,014)
   (VALPROC,EXTPROCS)
   (LABEL,TYPELAB,EXTIDLIST)
   (SWITCH,TYPESWITCH,EXTIDLIST)
   (TABLE,TYPEINT,TYPEARRAY,EXTID,OSB,INT,CSB,TABLEDETAIL)
   (SETYES,TYPEPROC,PROCEDURE,EXTPROCS)

014=(09,EXTIDLIST)
   (TYPETPROC,PROCEDURE,EXTPROCS)

09=()
   (ARRAY,TYPEARRAY)

EXTIDLIST=(EXTID,012)

EXTID=(ID,DIV,SETTEN,INT,ZINT,EXTADD,NEWNAME)

012=()
   (COMMA,EXTIDLIST)

EXTPROCS=(EXTID,PARAMLEV2,013)

013=()
   (COMMA,EXTPROCS)

015=()
   (SEMI,EXTLIST)

ABSLIST=(CLEARTYPE,ABSITEM,011)

ABSITEM=()
   (TYPE,010)
   (VALPROC,ABSPROCS)
   (LABEL,TYPELAB,ABSIDLIST)
   (SWITCH,TYPESWITCH,ABSIDLIST)
   (TABLE,TYPEINT,TYPEARRAY,ABSID,OSB,INT,CSB,TABLEDETAIL)
   (SETYES,TYPEPROC,PROCEDURE,ABSPROCS)

010=(09,ABSIDLIST)
   (TYPETPROC,PROCEDURE,ABSPROCS)

ABSIDLIST=(ABSID,07)

ABSID=(I),DIV,INT ABSADD,NEWNAME)
```



```
07=()
   (COMMA,ABSIDLIST)

ABSPROCS=(ABSID,PARAMLEV2,08)

08=()
   (COMMA,ABSPROCS)

011=()
   (SEMI,ABSLIST)

068=(COMMONDEC)
   (PROGRAM)
   (LIBRARY,CLEARTYPE,066)
```

```
PROCCALL=(SETUPPROC,SELPARAMS,CALLPROC,FINISHPROC)

PARAMLOOP=(NEXTPTYPE,SELPTYPE,ANYMORE,SELMORE)

MULTITEST=(CRB)
   (COMMA,CALLPROC,MULTICALL,PARAMLOOP)

'SEMANTIC' 'RULE'

PROCSEL=(VARCHECK)
   (PROCCALL)

SELPARAMS=()
   (ORB,PARAMLOOP)

SELPTYPE=(EXPR)
   (LOCVAR)
   (ID,LOOKUP)
   (ID,OSB)

SELMORE=(MULTITEST)
   (COMMA,PARAMLOOP)
```



```
NPART=(NREAD)
   (NWRITE)
   (NSHIFT)
   (NJUMP)
   (NLOC)

'SYNTAX' 'RULE'

NLOC=(NWRITE)
   (STRING,STRINGEX)

NJUMP=(ID,LABL)
   (SELF,ZINT,RELADD)

NSHIFT=(INT,CODESHIFT)
   (MODACC,ZINT,CODESHIFT,NISMOD)

NWRITE=(ANYACC,NISACC)
   (ID,LOOKUPD,NW1)
   (NONAME,NW2)

NW1=()
   (NW2)

NW2=(OSB,NW3,INCTOS,CSB)

NW3=(SGINT)
   (MODACC,NISMOD,ZINT)

NREAD=(NWRITE)
   (SGINT,CONSTANT)
   (CLEARTYPE,TYPE,ORB,PNUMBER,SCALECON,CRB,CONSTANT)
```



```
'SYNTAX' 'RULE'

A1=(BECOMES,VARCHECK,SETASS,A2,STORE,BENEAD)

'SEMANTIC' 'RULE'

A2=(A3)
   (EXPR)

'SYNTAX' 'RULE'

A3=(CEXPR)
   (STRING,STRINGEX)
   (ZERO,STOREZERO)
   (TERM,ADDA,A4)
   (PLUS,TERM,ADDA,A4)
   (MINUS,TERM,SUBA,A4)

A4=(SIMPLEASS)
   (SELOPTA,A5)

'SEMANTIC' 'RULE'

A5=(064)
   (AXPPA)

'SYNTAX' 'RULE'

FORSTAT=(FOR,STARTFOR,LOCVAR,CHECKCV,BECOMES,F1)

F1=(A3,ASSCV,F2)

F2=(F3)
   (WHILE,CONDITIONLIST,WHILEL,F3)
   (STEP,EXPR,STEPUNT,UNTIL,EXPR,STEPUNT,F3)

F3=(DO,DOCS,S,ENDFOR)
   (COMMA,MOREFOR,F1)

'FINISH'
```

```
A1              DECS(1),RULES 5(1),RULES 8(2),ASSIGNMENT(1),5;
A2              DECS(1),ASSIGNMENT(2),3;
A3              DECS(1),ASSIGNMENT(3),4;
A4              DECS(1),ASSIGNMENT(4),5;
A5              DECS(1),ASSIGNMENT(2),3;
APSADD          LAB1(1),RULES 10(1),2;
APSID           DECS(1),RULES 10(3),RULES 11(1),5;
APSIDLIST       DECS(1),RULES 10(4),RULES 11(1),6;
APSITEM         DECS(1),RULES 10(2),3;
APSLIST         DECS(1),RULES 1(1),RULES 10(1),RULES 11(1),4;
APSOLUTE        DECS(1),TS 2(1),RULES 1(1),3;
APSPROCS        DECS(1),RULES 10(3),RULES 11(2),6;
ADD             LAB1(1),RULES 7(1),2;
ADDA            LAB1(1),ASSIGNMENT(2),3;
ADDSPEC         LAB1(1),RULES 2(1),RULES 3(1),3;
ADDRSV          LAB1(1),RULES 4(1),2;
AND             DECS(1),TS 1(1),RULES 8(1),3;
APSCHECK        LAB1(1),RULES 8(1),2;
APSVEP          DECS(1),TS 2(1),RULES 8(1),3;
ARGLERSTAT      DECS(1),RULES 5(1),RULES 8(2),4;
ARYACC          DECS(1),TS 1(1),CODE(1),3;
ARYMORE         PROCS(1),PROC CALLS(1),2;
ARYPREF         LAB1(1),RULES 5(1),2;
ARRAY           DECS(1),TS 2(1),RULES 1(1),RULES 2(1),RULES 3(1),RULES 4(1),RULES 10(1),7;
ARRAYLIST       DECS(1),RULES 1(2),RULES 2(1),4;
ARRAYTAIL       DECS(1),RULES 2(3),4;
ASSCV           LAB1(1),ASSIGNMENT(1),2;
ASSIGNMENT      DECS(1),TS 1(1),RULES 9(2),4;
ASSTRACE        LAB1(1),RULES 9(2),5;
AXPR            DECS(1),RULES 6(3),4;
AXPRA           DECS(1),RULES 6(2),ASSIGNMENT(1),4;
AYNCHECK        LAB1(1),RULES 6(1),2;

BASE            DECS(1),RULES 3(2),RULES 4(1),4;
BECOMES         DECS(1),TS 2(1),RULES 2(1),RULES 3(1),RULES 4(1),RULES 10(1),ASSIGNMENT(2),8;
BEGIN           DECS(1),TS 2(1),RULES 4(1),RULES 5(3),RULES 8(1),7;
BEGINBLOCK      PROCS(1),RULES 5(1),RULES 8(1),3;
BEGINPROC       LAB1(1),RULES 4(2),RULES 9(1),4;
BEGINPROG       LAB1(1),RULES 4(1),2;
BEGINPSPEC      LAB1(1),RULES 2(1),2;
BEHEAD          PROCS(1),RULES 8(1),ASSIGNMENT(1),3;
BITS            DECS(1),TS 2(1),RULES 7(1),3;
BITSET          DECS(1),RULES 7(2),RULES 8(1),4;
BITSIN          LAB1(1),RULES 7(1),2;
BODY            DECS(1),RULES 4(2),RULES 5(1),4;
BOUNDPAIR       DECS(1),RULES 1(1),RULES 2(3),5;

CALLPROC        LAB1(1),PROC CALLS(2),3;
CEXPR           DECS(1),RULES 6(3),ASSIGNMENT(1),5;
CHECKCV         LAB1(1),ASSIGNMENT(1),2;
CLEARTYPE       LAB1(1),RULES 3(4),RULES 7(2),RULES 9(2),RULES 10(2),RULES 11(1),CODE(1),13;
CODE            DECS(1),TS 2(1),RULES 5(2),4;
CODELAB         LAB1(1),RULES 5(1),2;
CODESHIFT       LAB1(1),CODE(2),3;
COLIATION       DECS(1),RULES 6(1),RULES 7(2),4;
COLON           DECS(1),TS 1(1),RULES 2(1),RULES 3(1),RULES 4(1),RULES 5(2),RULES 8(1),RULES 10(1),9;
COMMA           DECS(1),TS 1(1) RULES 1(3),RULES 2(4),RULES 3(3),RULES 4(3),RULES 5(1),RULES 6(1),RULES 7(1),
                RULES 9(3),RULES 10(2),RULES 11(2),PROC CALLS(2),ASSIGNMENT(1),28;
COMMAB          DECS(1),TS 2(1),RULES 3(1),3;
COMMON          DECS(1),TS 2(1),RULES 1(1),3;
```



```
COMMONDEC       DECS(1),RULES 1(2),RULES 11(1),4;
COMMONITEM      DECS(1),RULES 1(2),3;
COMMONLIST      DECS(1),RULES 1(2),RULES 4(1),4;
COMPEF          LAB1(1),RULES 1(1),2;
COMON           LAB1(1),RULES 1(1),2;
COMPILE         DECS(1),TS 2(1),2;
COMPILEITEM     DECS(1),RULES 1(2),3;
COMPOUND        DECS(1),RULES 5(3),4;
COMPROCS        DECS(1),RULES 1(3),RULES 2(1),RULES 3(1),6;
CONSPECID       DECS(1),RULES 1(1),RULES 2(2),4;
CONSPECIDCONT   DECS(1),RULES 1(2),3;
CONSPECIDLIST   DECS(1),RULES 1(4),5;
CONDADD         LAB1(1),RULES 8(1),2;
CONDITION       DECS(1),RULES 5(2),3;
CONDITIONLIST   DECS(1),RULES 5(2),RULES 6(1),RULES 8(2),ASSIGNMENT(1),7;
CONDMINUS       LAB1(1),RULES 7(1),2;
CONDPLUS        LAB1(1),RULES 7(2),3;
CONDSUB         LAB1(1),RULES 8(1),2;
CONDTERM        DECS(1),RULES 7(2),3;
CONDTERMS       DECS(1),RULES 7(1),RULES 8(3),5;
CONSTANT        LAB1(1),RULES 6(2),RULES 7(2),CODE(2),7;
CRB             DECS(1),TS 1(1),RULES 1(3),RULES 2(2),RULES 4(2),RULES 7(5),RULES 9(1),PROC CALLS(1),CODE(1),17;
CRBCSB          DECS(1),TS 1(1),RULES 1(1),RULES 3(1),4;
CS              DECS(1),RULES 5(3),4;
CSB             DECS(1),TS 2(1),RULES 1(2),RULES 3(4),RULES 4(1),RULES 6(2),RULES 7(1),RULES 10(2),CODE(1),15;
CSL             DECS(1),RULES 5(2),RULES 8(3),6;

D               DECS(1),RULES 4(2),3;
DATADEC         DECS(1),RULES 3(2),3;
DECSIZE         LAB1(1),RULES 1(1),2;
DIFFER          DECS(1),TS 1(1),RULES 7(1),3;
DIFFERENCE      DECS(1),RULES 6(3),4;
DIV             DECS(1),TS 1(1),RULES 3(1),RULES 7(1),RULES 9(1),RULES 10(2),7;
DIVIDE          LAB1(1),RULES 7(1),2;
DL              DECS(1),RULES 4(2),RULES 5(1),RULES 8(1),RULES 9(1),6;
DO              DECS(1),TS 2(1),ASSIGNMENT(1),3;
DOCS            LAB1(1),ASSIGNMENT(1),2;
DOSHIFT         LAB1(1),RULES 7(1),2;
DUMP            DECS(1),TS 2(1),2;

ELSE            DECS(1),TS 2(1),RULES 5(1),RULES 6(1),4;
ELSEFI          LAB1(1),RULES 6(2),3;
ELSES           LAB1(1),RULES 5(1),2;
ELSEX           LAB1(1),RULES 6(1),2;
END             DECS(1),TS 1(1),RULES 4(1),RULES 5(3),RULES 8(3),9;
ENDARRAY        LAB1(1),RULES 1(1),2;
ENDBLOCK        PROCS(1),RULES 5(1),RULES 8(1),3;
ENDDECS         LAB1(1),RULES 4(1),RULES 5(2),RULES 8(2),6;
ENDFOR          LAB1(1),ASSIGNMENT(1),2;
ENDPROC         LAB1(1),RULES 4(1),2;
ENDPROG         LAB1(1),RULES 4(1),2;
ENDPSPEC        LAB1(1),RULES 2(2),3;
ENDST           LAB1(1),RULES 5(2),3;
ENTERPROC       PROCS(1),RULES 5(1),2;
EXITCHECK       LAB1(1),RULES 8(2),3;
EXPR            DECS(1),RULES 6(3),RULES 7(2),RULES 8(1),PROC CALLS(1),ASSIGNMENT(3),11;
EXPRTYPE        LAB1(1),RULES 7(1),2;
EXTADD          LAB1(1),RULES 10(1),2;
EXTERNAL        DECS(1),TS 2(1),RULES 1(1),3;
EXTID           DECS(1),RULES 10(4),5;
```

```
EXTYPELIST      DECS(1),RULES 10(5),6;
EXTYITEM        DECS(1),RULES 10(2),3;
EXTYLIST        DECS(1),RULES 1(1),RULES 10(2),4;
EXTYPROCS       DECS(1),RULES 10(5),6;

F1              DECS(1),ASSIGNMENT(3),4;
F2              DECS(1),ASSIGNMENT(2),3;
F3              DECS(1),ASSIGNMENT(4),5;
FACTOR          DECS(1),RULES 6(2),RULES 7(2),5;
FI              LAB1(1),RULES 5(1),2;
FIELDDISP       LAB1(1),RULES 3(2),3;
FIELDLIST       DECS(1),RULES 3(3),4;
FIELDPOSN       LAB1(1),RULES 3(1),RULES 7(1),3;
FIELDTYPE       DECS(1),RULES 3(2),3;
FILE            LAB1(1),RULES 6(1),2;
FINISH          DECS(1),TS 2(1),RULES 1(1),3;
FINISHPROC      LAB1(1),PROC CALLS(1),2;
FINISHSEG       PROCS(1),RULES 10(3),4;
FIRSTDIM        LAB1(1),RULES 2(1),2;
FIXED           DECS(1),TS 2(1),RULES 1(1),3;
FLOATING        DECS(1),TS 2(1),RULES 1(1),3;
FOR             DECS(1),TS 2(1),ASSIGNMENT(1),3;
FORSTAT         DECS(1),RULES 5(1),RULES 8(1),ASSIGNMENT(1),4;
FORTRACE        LAB1(1),RULES 9(2),3;

GOTO            DECS(1),TS 2(1),RULES 8(1),3;
GOTOL           LAB1(1),RULES 8(1),2;
GOTOSK          LAB1(1),RULES 8(1),2;
GOTOSTAT        DECS(1),RULES 5(1),RULES 8(2),4;

IN              DECS(1),TS 2(1),RULES 1(1),RULES 3(3),RULES 4(1),RULES 5(2),RULES 6(1),RULES 7(2),RULES 8(3),
                RULES 9(2),RULES 10(2),PROC CALLS(2),CODE(2),23;
INITST          DECS(1),RULES 1(4),RULES 4(4),RULES 5(1),10;
IF              DECS(1),TS 1(1),RULES 5(1),RULES 6(1),4;
IFEX            LAB1(1),RULES 6(1),2;
IFS             LAB1(1),RULES 5(1),2;
IFSTAT          DECS(1),RULES 5(2),RULES 8(1),4;
INCYOS          LAB2(1),RULES 3(2),RULES 4(1),CODE(1),5;
INST            DECS(1),TS 1(1),RULES 5(1),3;
INSTTYPE        PROCS(1),RULES 5(1),2;
INT             DECS(1),TS 1(1),RULES 1(5),RULES 3(4),RULES 4(1),RULES 7(2),RULES 9(2),RULES 10(4),CODE(1),21;
INTCON          DECS(1),TS 1(1),RULES 7(1),3;
INTEGER         DECS(1),TS 2(1),RULES 1(1),3;
INTFIXFIELD     DECS(1),RULES 3(2),3;

KILLEXPR        PROCS(1),RULES 6(2),3;
KILLPARAMS      LAB2(1),RULES 5(1),2;

LABEL           DECS(1),TS 2(1),RULES 1(1),RULES 2(1),RULES 4(1),RULES 7(1),RULES 9(2),RULES 10(2),10;
LABL            LAB2(1),RULES 8(1),CODE(1),3;
LABSK           LAB2(1),RULES 8(1),RULES 8(1),3;
LABTRACE        LAB2(1),RULES 9(2),3;
LASTDIM         LAB2(1),RULES 2(1),2;
LCOND           DECS(1),RULES 5(1),RULES 6(1),3;
LEVEL           DECS(1),TS 2(1),RULES 9(1),3;
LEVELDEC        DECS(1),RULES 1(1),RULES 4(1),RULES 9(1),4;
LHSB            DECS(1),RULES 8(2),3;
LHSBITS         LAB2(1),RULES 8(2),3;
LHSPROC         LAB2(1),RULES 8(1),2;
LIBADD          LAB2(1),RULES 9(1),2;
```



```
LIBID           DECS(1),RULES 9(3),RULES 10(4),8;
LIBIDLIST       DECS(1),RULES 9(3),4;
LIBITEM         DECS(1),RULES 9(2),3;
LIBLIST         DECS(1),RULES 9(3),4;
LIBPROCS        DECS(1),RULES 9(5),6;
LIBRARY         DECS(1),TS 2(1),RULES 1(1),RULES 11(1),4;
LOAD            DECS(1),TS 2(1),2;
LOCACT          PROCS(1),RULES 7(1),2;
LOCATION        DECS(1),TS 1(1),RULES 2(1),RULES 4(1),RULES 7(1),5;
LOCVAR          DECS(1),RULES 6(1),RULES 7(1),PROC CALLS(1),ASSIGNMENT(1),5;
LOCVUP          PROCS(1),RULES 6(2),RULES 7(1),RULES 8(6),PROC CALLS(1),9;
LOCVUPD         LAB2(1),RULES 3(1),CODE(1),3;
LOOP            DECS(1),TS 1(1),RULES 9(2),4;

MASK            DECS(1),TS 1(1),RULES 7(1),3;
MINAI"          LAB2(1),RULES 2(1),2;
MIN'S           DECS(1),TS 1(1),RULES 1(1),RULES 2(1),RULES 6(2),RULES 7(2),RULES 8(1),ASSIGNMENT(1),10;
MINISSUB        LAB2(1),RULES 6(1),2;
MODACC          DECS(1),TS 1(1),CODE(2),4;
MORROUNDS       DECS(1),RULES 1(1),RULES 2(1),3;
MORROR          LAB2(1),ASSIGNMENT(1),2;
MORRSPEC        DECS(1),RULES 3(1),RULES 4(2),4;
MORRSUBS        DECS(1),RULES 6(3),4;
MOV             LAB2(1),RULES 7(1),2;
MSK             LAB2(1),RULES 7(1),2;
MULT            DECS(1),TS 1(1),RULES 7(1),3;
MULTICALL       LAB2(1),PROC CALLS(1),2;
MULTITEST       DECS(1),PROC CALLS(2),3;

NEGNOT          LAB2(1),RULES 1(1),RULES 2(1),3;
NEQ             LAB2(1),RULES 7(1),2;
NEWID           DECS(1),RULES 1(2),RULES 2(1),RULES 3(2),RULES 4(6),RULES 5(2),RULES 9(1),RULES 10(1),16;
NEWLABLE        LAB2(1),RULES 1(1),RULES 3(1),RULES 9(1),RULES 10(2),6;
NEWPROC         PROCS(1),RULES 2(3),4;
NEXTBSET        LAB2(1),RULES 3(1),RULES 4(1),RULES 5(1),RULES 10(1),5;
NEXTBTYPE       PROCS(1),PROC CALLS(1),2;
NISACC          LAB2(1),CODE(1),2;
NISMOD          LAB2(1),CODE(2),3;
NJUMP           DECS(1),CODE(2),3;
NLOOP           DECS(1),CODE(2),3;
NOT             DECS(1),TS 1(1),RULES 6(1),RULES 5(1),6;
NOAFTER         LAB2(1),RULES 1(1),RULES 3(1),3;
NOBITS          LAB2(1),RULES 1(2),RULES 3(1),RULES 7(1),5;
NONAME          PROCS(1),RULES 3(1),RULES 4(1),RULES 8(1),RULES 7(1),RULES 8(2),CODE(1),8;
NOSIG           LAB2(1),RULES 3(1),RULES 7(1),3;
NOWAIT          DECS(1),RULES 5(1),CODE(1),3;
NREAD           DECS(1),CODE(2),3;
NREWRT          DECS(1),CODE(2),3;
NSET            DECS(1),CODE(2),3;
NTEST           DECS(1),CODE(3),4;
NUP             DECS(1),CODE(2),3;
NWRITE          DECS(1),CODE(4),5;

OR              DECS(1),RULES 7(6),5;
OS0             DECS(1),RULES 10(2),3;
OS1             DECS(1),RULES 10(1),RULES 11(1),3;
OS2             DECS(1),RULES 10(2),3;
OS3             DECS(1),RULES 10(2),3;
OS4             DECS(1),RULES 10(2),3;
OS5             DECS(1),RULES 10(2),3;
```

```
016             DECS(1),RULES 9(2),3;
017             DECS(1),RULES 9(2),3;
018             DECS(1),RULES 9(2),3;
019             DECS(1),RULES 9(2),3;
02              DECS(1),RULES 7(3),4;
020             DECS(1),RULES 2(1),RULES 3(1),3;
021             DECS(1),RULES 1(2),RULES 3(1),RULES 4(1),5;
022             DECS(1),RULES 1(2),3;
023             DECS(1),RULES 1(1),RULES 4(1),3;
024             DECS(1),RULES 6(3),4;
025             DECS(1),RULES 8(3),4;
026             DECS(1),RULES 5(1),RULES 8(2),4;
027             DECS(1),RULES 8(2),3;
028             DECS(1),RULES 7(2),3;
029             DECS(1),RULES 6(2),3;
03              DECS(1),RULES 6(1),RULES 7(2),4;
030             DECS(1),RULES 7(2),3;
031             DECS(1),RULES 5(1),RULES 8(1),3;
032             DECS(1),RULES 6(1),RULES 7(1),PROC CALLS(1),4;
033             DECS(1),RULES 8(3),4;
034             DECS(1),RULES 5(1),RULES 8(1),3;
035             DECS(1),RULES 5(1),RULES 8(1),3;
037             DECS(1),RULES 5(2),3;
038             DECS(1),RULES 5(1),RULES 8(1),3;
039             DECS(1),RULES 5(1),RULES 8(1),3;
04              DECS(1),RULES 6(1),RULES 7(2),4;
040             DECS(1),RULES 2(2),3;
041             DECS(1),RULES 9(2),3;
042             DECS(1),RULES 1(1),RULES 2(1),3;
043             DECS(1),RULES 3(2),3;
044             DECS(1),RULES 3(1),RULES 4(1),3;
045             DECS(1),RULES 3(2),3;
046             DECS(1),RULES 3(1),RULES 4(1),3;
047             DECS(1),RULES 4(2),3;
048             DECS(1),RULES 4(1),RULES 9(1),3;
049             DECS(1),RULES 9(2),3;
05              DECS(1),RULES 6(1),RULES 7(3),5;
050             DECS(1),RULES 4(1),RULES 9(1),3;
051             DECS(1),RULES 5(1),RULES 8(1),3;
052             DECS(1),RULES 1(2),3;
053             DECS(1),RULES 2(2),3;
054             DECS(1),RULES 2(2),3;
055             DECS(1),RULES 2(2),3;
056             DECS(1),RULES 2(1),RULES 3(1),3;
057             DECS(1),RULES 2(2),3;
058             DECS(1),RULES 4(2),3;
059             DECS(1),RULES 1(2),3;
06              DECS(1),RULES 6(3),RULES 7(3),7;
060             DECS(1),RULES 3(3),4;
060A            DECS(1),RULES 3(2),3;
061             DECS(1),RULES 5(2),3;
062             DECS(1),RULES 4(2),3;
063             DECS(1),RULES 4(2),3;
064             DECS(1),RULES 4(1),RULES 5(1),3;
065             DECS(1),RULES 4(1),RULES 5(1),3;
066             DECS(1),RULES 9(1),RULES 10(1),RULES 11(1),4;
067             DECS(1),RULES 10(2),3;
068             DECS(1),RULES 1(1),RULES 11(1),3;
069             DECS(1),RULES 1(1),RULES 9(1),3;
064             DECS(1),RULES 7(2),ASSIGNMENT(1),4;
```



```
07              DECS(1),RULES 10(1),RULES 11(1),3;
070             DECS(1),RULES 1(2),3;
08              DECS(1),RULES 11(2),3;
09              DECS(1),RULES 10(3),4;
0bL             DECS(1),RULES 5(2),RULES 8(1),4;
0FFSTEXPR       LAB2(1),RULES 7(1),2;
0NERIT          LAB2(1),RULES 7(1),2;
0NERIT"         LAB2(1),RULES 2(1),2;
0NFORMORE       DECS(1),RULES 7(2),3;
0R              DECS(1),TS 1(1),RULES 8(1),3;
0RACT           LAB2(1),RULES 8(1),2;
0sN             DECS(1),TS 1(1),RULES 1(3),RULES 2(2),RULES 3(1),RULES 4(1),RULES 7(5),RULES 9(1),PROC CALLS(1),
                CODE(1),17;
3RBOSB          DECS(1),TS 1(1),RULES 1(1),RULES 3(1),4;
0sF             LAB2(1),RULES 7(1),2;
0SB             DECS(1),TS 2(1),RULES 1(2),RULES 3(4),RULES 4(1),RULES 6(1),RULES 7(1),RULES 10(2),CODE(1),14;
0UTCJ           PROCS(1),RULES 8(1),2;
0UTCODE         LAB2(1),RULES 5(1),2;
0UTPRESET       LAB2(1),RULES 2(1),2;
0VERFLOW        DECS(1),TS 1(1),RULES 5(1),3;
0VERLAY         DECS(1),TS 2(1),RULES 3(1),3;
0VERLAYDEC      DECS(1),RULES 1(1),RULES 3(1),RULES 4(1),4;
0VEROFF         LAB2(1),RULES 3(1),2;
0VERON          LAB2(1),RULES 3(1),2;
0VBYTEST        LAB2(1),RULES 5(1),2;

PAIRLIST        DECS(1),RULES 4(1),RULES 5(3),5;
PARAMETERPART   DECS(1),RULES 4(2),3;
PARAMETERS      DECS(1),RULES 4(2),RULES 5(1),4;
PARAMETERSET    DECS(1),RULES 4(2),3;
PARAMLEV2       DECS(1),RULES 2(2),RULES 4(1),RULES 9(1),RULES 10(1),RULES 11(1),7;
PARAMLOOP       DECS(1),PROC CALLS(4),5;
PARAMTAB        LAB2(1),RULES 4(2),3;
PARAMTYPE       DECS(1),RULES 2(2),3;
PARAMTYPELIST   DECS(1),RULES 2(2),RULES 3(1),4;
PARTFIELD       DECS(1),RULES 3(3),4;
PARTINT         LAB2(1),RULES 1(1),RULES 3(1),RULES 7(1),4;
PCONST          DECS(1),TS 1(1),RULES 2(3),5;
PLUS            DECS(1),TS 1(1),RULES 1(1),RULES 2(1),RULES 6(2),RULES 7(2),RULES 8(1),ASSIGNMENT(1),10;
PLUSSUB         LAB2(1),RULES 6(3),RULES 7(2),6;
PNUMBER         DECS(1),RULES 2(2),RULES 3(1),RULES 4(1),RULES 10(1),CODE(1),7;
POWER           DECS(1),TS 2(1),RULES 6(1),3;
PRESET          DECS(1),TS 2(1),2;
PRESETLIST      DECS(1),RULES 1(2),RULES 2(1),RULES 4(2),6;
PRESETNUM       DECS(1),RULES 2(2),3;
PRESETS         DECS(1),RULES 2(4),5;
PRESETSTRING    LAB2(1),RULES 2(1),2;
PRT             DECS(1),RULES 7(3),4;
PROCBODY        DECS(1),RULES 4(1),RULES 5(1),3;
PROCCALL        DECS(1),RULES 8(1),PROC CALLS(2),4;
PROCEDURE       DECS(1),TS 2(1),RULES 1(2),RULES 2(3),RULES 3(1),RULES 4(3),RULES 5(1),RULES 9(4),RULES 10(6),22;
PROCENTRY       LAB2(1),RULES 8(2),3;
PROCPARAMLIST   DECS(1),RULES 4(4),RULES 5(1),6;
PROCREST        DECS(1),RULES 4(3),RULES 9(1),RULES 10(3),8;
PROCTRACE       LAB2(1),RULES 9(2),3;
PROGRAM         DECS(1),RULES 1(1),RULES 4(1),RULES 11(1),4;

RAISE           LAB2(1),RULES 8(1),2;
RCMND           DECS(1),RULES 5(1),RULES 7(1),3;
REALCON         DECS(1),TS 1(1),RULES 7(1),3;
```

```
$PCOVER         DECS(1),TS 2(1),2;
$PLADD          LAB2(1),CODE(1),2;
$PLATION        LAB2(1),RULES 5(1),2;
$NSBITS         LAB2(1),RULES 7(1),2;
$NSPSEL         DECS(1),RULES 7(1),PROC CALLS(1),3;
$NSPTEST        PROCS(1),RULES 7(1),2;
$N              DECS(1),TS 1(1),RULES 7(1),3;
$NN             DECS(1),RULES 1(2),3;

$               DECS(1),RULES 5(3),RULES 8(1),ASSIGNMENT(1),6;
$ALECON         PROCS(1),RULES 10(1),CODE(1),3;
$ALETERM        LAB2(1),RULES 8(3),RULES 7(1),5;
$PC             DECS(1),RULES 7(3),4;
$PLF            DECS(1),TS 1(1),RULES 3(1),CODE(1),4;
$PLMORE         DECS(1),PROC CALLS(2),3;
$PLOPTA         PROCS(1),ASSIGNMENT(1),2;
$PLPARAMS       DECS(1),PROC CALLS(2),3;
$PLPTYPE        DECS(1),PROC CALLS(2),3;
$PMI            DECS(1),TS 1(1),RULES 1(1),RULES 3(1),RULES 4(3),RULES 5(1),RULES 8(2),RULES 9(1),RULES 10(1),
                RULES 11(1),13;
$PTANS          LAB2(1),RULES 8(1),2;
$PTASS          PROCS(1),ASSIGNMENT(1),2;
$PTLAB          LAB2(1),RULES 8(1),2;
$PTLB           LAB2(1),RULES 2(1),2;
$PTLEVEL        LAB2(1),RULES 9(1),2;
$PTLIBSEG       LAB2(1),RULES 10(3),4;
$PTLIBVAR       LAB2(1),RULES 10(1),2;
$PTNEZ          LAB3(1),RULES 7(1),2;
$PTNO           LAB3(1),RULES 4(1),RULES 5(1),3;
$PTPARAMS       LAB3(1),RULES 5(1),2;
$PTPREF         LAB3(1),RULES 7(1),2;
$PTSHIFT        LAB3(1),RULES 7(1),2;
$PTTEN          LAB3(1),RULES 3(1),RULES 10(1),3;
$PTTEST         LAB3(1),RULES 1(1),2;
$PTUB           LAB3(1),RULES 2(1),2;
$PTUPPROC       PROCS(1),PROC CALLS(1),2;
$PTUPSUB        LAB3(1),RULES 6(1),2;
$PTYES          LAB3(1),RULES 1(1),RULES 2(1),RULES 4(2),RULES 5(1),RULES 9(1),RULES 10(3),10;
$QINT           DECS(1),RULES 1(2),RULES 2(2),RULES 3(6),RULES 4(1),CODE(2),14;
$RIPT           DECS(1),TS 1(1),RULES 7(1),3;
$RIPTS          DECS(1),RULES 7(2),3;
$IMPLEASS       LAB3(1),ASSIGNMENT(1),2;
$INGLESUBS      DECS(1),RULES 6(3),RULES 7(1),RULES 8(3),8;
$KIPDTA         LAB3(1),RULES 2(1),2;
$L              DECS(1),RULES 4(1),RULES 5(2),RULES 8(2),6;
$PECADD         DECS(1),RULES 3(2),3;
$PECCON         LAB3(1),RULES 4(2),3;
$PECIAL         DECS(1),TS 1(1),RULES 3(1),3;
$PECIALDEC      DECS(1),RULES 1(1),RULES 3(1),RULES 4(1),4;
$PECITEM        DECS(1),RULES 3(2),3;
$PECLAB         LAB3(1),RULES 3(1),RULES 9(1),3;
$PECLIST        DECS(1),RULES 3(2),RULES 4(1),4;
$PECNUM         LAB3(1),RULES 3(2),RULES 4(1),4;
$PECONE         LAB3(1),RULES 2(1),2;
$PECREL         LAB3(1),RULES 3(1),2;
$PECSTRING      PROCS(1),RULES 3(1),2;
$S              DECS(1),RULES 8(2),3;
$T              DECS(1),RULES 5(3),4;
$TACKEXPR       PROCS(1),RULES 5(1),RULES 7(1),3;
$TARTDEC        LAB3(1),RULES 1(1),RULES 4(1),3;
```



```
$TARTFOR        LAB3(1),ASSIGNMENT(1),2;
$TATUSCHECK     PROCS(1),RULES 5(2),RULES 8(5),8;
$TATUSCODE      PROCS(1),RULES 5(2),3;
$TEP            DECS(1),TS 2(1),ASSIGNMENT(1),3;
$TEPUNT         LAB3(1),ASSIGNMENT(2),3;
$TORE           LAB3(1),ASSIGNMENT(1),2;
$TOREZERO       LAB3(1),ASSIGNMENT(1),2;
$TRING          DECS(1),TS 1(1),RULES 2(1),RULES 3(1),RULES 6(1),CODE(1),ASSIGNMENT(1),7;
$TRINGEX        LAB3(1),RULES 6(1),CODE(1),ASSIGNMENT(1),4;
$UB             LAB3(1),RULES 7(1),2;
$UBA            LAB3(1),ASSIGNMENT(1),2;
$UBCOMMA        LAB3(1),RULES 6(1),2;
$UBEX           DECS(1),RULES 6(3),RULES 7(1),5;
$UBEXCONT       DECS(1),RULES 6(4),5;
$UBSCRIPT       DECS(1),RULES 6(2),RULES 7(1),RULES 8(1),5;
$UBSTART        DECS(1),RULES 6(3),4;
$UBTERMS        DECS(1),RULES 6(3),4;
$WITCH          DECS(1),TS 2(1),RULES 1(1),RULES 2(1),RULES 4(2),RULES 10(2),8;
$WLIST          DECS(1),RULES 4(1),RULES 9(2),4;

$ABADD          LAB3(1),RULES 3(1),2;
$ABLE           DECS(1),TS 2(1),RULES 2(1),RULES 3(1),RULES 4(1),RULES 10(2),7;
$ABLEDEC        DECS(1),RULES 1(1),RULES 3(2),RULES 4(1),5;
$ABLEDETAIL     DECS(1),RULES 3(2),RULES 4(1),RULES 10(2),6;
$ABLESIZE       LAB3(1),RULES 3(1),2;
$EMP            DECS(1),RULES 6(7),RULES 7(5),RULES 8(2),ASSIGNMENT(3),10;
$ERM            DECS(1),RULES 7(3),4;
$EST            DECS(1),TS 2(1),RULES 1(1),3;
$HEN            DECS(1),TS 1(1),RULES 5(1),RULES 6(1),4;
$HENEX          LAB3(1),RULES 6(1),2;
$RI             DECS(1),RULES 7(3),4;
$RACE           DECS(1),TS 1(1),RULES 1(1),RULES 4(1),RULES 9(4),8;
$RACETYPE       DECS(1),RULES 1(1),RULES 4(1),RULES 9(1),4;
$YPE            DECS(1),RULES 1(2),RULES 2(3),RULES 3(3),RULES 4(3),RULES 5(1),RULES 7(1),RULES 9(1),RULES 10(3),
                CODE(1),19;
$YPEARRAY       LAB3(1),RULES 1(1),RULES 3(3),RULES 10(3),8;
$YPEFLOAT       LAB3(1),RULES 1(1),2;
$YPEIARRAY      LAB3(1),RULES 2(2),RULES 4(2),5;
$YPEINT         LAB3(1),RULES 1(1),RULES 2(1),RULES 3(4),RULES 4(1),RULES 10(2),10;
$YPESWITCH      LAB3(1),RULES 1(1),RULES 2(1),RULES 4(1),4;
$YPELAB         LAB3(1),RULES 1(1),RULES 2(1),RULES 4(1),RULES 10(2),6;
$YPELOC         LAB3(1),RULES 2(1),RULES 4 1),3;
$YPEPROC        LAB3(1),RULES 1(1),RULES 2(1),RULES 4(2),RULES 5(1),RULES 9(1),RULES 10(3),10;
$YPESPEC        LAB3(1),RULES 2(2),RULES 4(1),RULES 5(1),5;
$YPESWITCH      LAB3(1),RULES 4(1),RULES 10(2),4;
$YPETPROC       LAB3(1),RULES 1(1),RULES 2(1),RULES 4(2),RULES 9(1),RULES 10(3),9;
$YPEVPROC       LAB3(1),RULES 2(1),RULES 3(1),RULES 5(1),4;
$YPEXPR         LAB3(1),RULES 7(1),2;

UNARYMINUS      PROCS(1),RULES 6(1),2;
UNION           DECS(1),TS 1(1),RULES 7(1),3;
UNTIED          DECS(1),RULES 6(2),RULES 7(1),4;
UNSPECLB        LAB3(1),RULES 3(1),RULES 7(1),3;
UNSTACKED       DECS(1),TS 2(1),RULES 3(1),3;
UNTIL           DECS(1),TS 2(1),ASSIGNMENT(1),3;

VALPROC         DECS(1),RULES 1(1),RULES 3(1),RULES 4(1),RULES 9(1),RULES 10(1),8;
VALUE           DECS(1),TS 2(1),RULES 2(1),RULES 3(1),RULES 4(1),5;
WARDCHECK       LAB3(1),RULES 7(1),PROC CALLS(1),ASSIGNMENT(1),4;
```

IDENTIFIER OCCURENCES IN SYNTAX 13


```
WHILE          DECS(1),TS 2(1),ASSIGNMENT(1),3;
WHILEL         LAB3(1),ASSIGNMENT(1),2;
WITH           DECS(1),TS 2(1),RULES 3(1),3;

YESNO          DECS(1),RULES 1(1),RULES 5(2),4;

ZERO           DECS(1),TS 1(1),RULES 6(3),RULES 7(1),ASSIGNMENT(1),7;
ZERONARRAYS    LAB3(1),RULES 1(1),2;
ZEROCOMP       LAB3(1),RULES ;(1),2;
ZERONUM        LAB3(1),RULES 2(1),RULES 3(1),3;
ZINT           DECS(1),RULES 3(2),RULES 10(1),CODE(3),7;
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference TN 799	3. Agency Reference	4. Report Security Classification UNCLASSIFIED
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location RSRE (MALVERN)		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title The Coral 66 Compiler for the Ferranti Argus 500 Computer

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
GORMAN, B				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

Computer Languages
Coral 66
Compiler

continue on separate piece of paper

Abstract

EN
DATE
FILME
DTIC