ELLIOTT 903 ALGOL

Translator store map. December 1966

8 - 5969	Translator and constants
5970 - 5999	Patch space
6000 - 6014	Workspace W
6015 - 6056	Input buffer INBUF (INBUF-1 is referenced)
6057 - 6146	Stack (See location SP + 1)
6147 - 7794	CODL growing upwards towards 8191 Namelist growing downwards towards 0
7795 - 7998	Built in names
7999	Spare
8000 - 8179	Spare

D. Hunter.

ELLIOTT 903 ALGOL

How to add names to the built in namelist December 1966

1. General

1.1 It does not matter where the name is added to the list which is in alphabetical order except that CHECKB, CHECKI and CHECKR are at the end; it was at one time necessary for them to be at the end, but this is no longer so.

1.2 If the namelist is altered in length the following changes must be made in the Translator on the assumption that the last name continues to occupy locations 7995 - 7998 inclusive.

1.2.1 The SIR directive at the front of the namelist must be reduced appropriately, e.g. by 8 to ↟ + 7787, for one extra procedure name with a few parameters.

1.2.2 At START + 9 the instruction 2 + 7795 must be changed to, e.g. 2 + 7787, for one extra procedure name. If this is not done the name will be cleared to zero at the start.

1.2.3 At START + 45 the instruction 4 - 200 must be changed to, e.g. 4 - 208 for one extra procedure name. If this is not done the name will not have its "used" bit set to zero at the start.

2. Structure of a namelist entry

Only procedure names are considered here. If a procedure has parameters then extra space is required for their codewords e.g. the procedures P, CAT and FRED with 0,. 1 and 5 parameters:-

Without parameters	One parameter	Five parameters
		+ 0 ⎫ + 0 ⎪ + 0 ⎪ codewords p 5 ⎭
	+ 0 ⎫ + 0 ⎪ + 0 ⎪ codewords p 1 ⎭	p 4 ⎫ p 3 ⎪ p 2 ⎪ codewords p 1 ⎭
£P	£CAT	£FRE
+ 0	+ 0	£D
wd 3	wd 3	wd 3
wd 4	wd 4	wd 4

It can be seen that although P occupies a single 4 word entry CAT requires two and FRED requires three entries of 4 words each.

Parameters, if present, are described by codewords p1, p2 etc. where p1 describes the first parameter, p2 the second etc. Up to 4 parameters can be accommodated in a block of four words. Codewords are as follows:-

Parameter type	Codeword	
real	&106100	If called by value add the constant &2000000
integer	&106500	
boolean	&105100	
real array	&046100	
integer array	&046500	
boolean array	&045100	
real procedure	&116120	Procedures without parameters. For procedures with parameters see below*
integer procedure	&116520	
boolean procedure	&115120	
procedure	&100120	
switch	&040200	
label	&100200	If called by value see above
string	&000040	

The block of 4 words holding the name and its details has the first six characters of the name stored left justified and space filled in words 1 and 2.

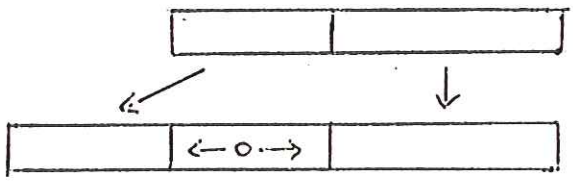
The third word wd 3 contains in its function part a 1 or a 2. If it contains a 1 then the procedure must be on the library tape and the address part of wd 3 must point to the name e.g.

1 ; - 2

One can arrange a shared group of names, see for example SIN, COS and ARCTAN sharing the name QATRIG. If this is done then the shared name must not start a block of four parameter entries. If it does then the "reset names" part of initialisation will treat the block as a name and reset the used bit to zero, spuriously.

If wd 3 contains a 2 in its function part then the 13 bit address part is treated as follows:-

The five most significant digits form the function part of a pord and the eight least significant digits form the address part with zeros between the functions and address parts.



The function part is usually /15 or 15 for a PRIM or INOUT operation. If an INOUT operation is involved and the procedure name is discovered to be inside a print or read list, eleven is added to the address part. In theory one could generate any pord one likes. If a PRIM operation is involved and the procedure is a type procedure the PRIM UP which is normally

produced is suppressed as is required by the interpreter.

The final word, wd 4, is constructed as follows:-

real procedure	without parameters	&116120	
integer procedure	without parameters	&116520	
boolean procedure	without parameters	&115120	
procedure	without parameters	&100120	
real procedure	with parameters	&036100	plus the number of parameters
integer procedure	with parameters	&036500	plus the number of parameters
boolean procedure	with parameters	&035100	plus the number of parameters
procedure	with parameters	&020100	plus the number of parameters

E.g. a real procedure with ten parameters would be &036112

* The codeword for a real procedure with parameters is &036100; note that the number of parameters is left blank in the codeword.

3. Assembly of the namelist

Use SIR at 512 to assemble the namelist. Then dump this by using the non-standard T22/23 to be found on the "USEFUL TAPES" tape and an appropriate data tape.

4. Adding namelist to translator

After input of the relocatable translator tape clear out the loader by using "CLEAR FROM 7168" on the "USEFUL TAPES" tape. Then dump the store from 8-7999 inclusive using the non-standard T22/23.

D. Hunter

CONTENTS

Routines continued

GENERAL STRATEGY OF THE TRANSLATOR

1. INTRODUCTION

The task of the Translator is to convert the ALGOL text into object program operations, which are assembled into object store by means of a loader/assembler, and obeyed interpretively at run time under the control of an Object Interpreter.

Lack of space necessitates the translator for 903 ALGOL to be one-pass, and object program operations are output as the source program is read in. Because insufficient store is available to hold the Translator, the translator Name List and subsidiary tables, and the object program, some such decision is imperative.

The object program is essentially a form of "Reverse Polish" notation, and the Translator uses a stack to perform the necessary re-ordering of the ALGOL symbols. The Translator also uses a "Name List" which holds details of the declaration and use of the various identifiers. During translation a great many checks are performed on the legality of the ALGOL text, but it cannot be claimed that these are exhaustive.

The heart of the Translator is a routine called the "Basic Cycle Routine" (BCR) that extracts the next section of ALGOL text (a section being a string of characters ending with a delimiter such as ";" or begin) ; control is then passed to a routine dealing with the delimiter concerned, and these routines may call further subroutines.

2. LIMITATIONS IMPOSED ON IFIP subset ALGOL

IFIP subset ALGOL restricts "full" ALGOL in several important ways (one of these being the exclusion of recursion). 903 ALGOL has further restricted IFIP subset ALGOL, in particular in the following two areas:

2.1 All identifiers must be declared before they are allowed to appear in expressions or statements. This simplifies a one-pass Translator's task considerably, as all relevant information about an identifier is in hand before it is actually used in processing.

This rule also applies to labels, which in 903 ALGOL must be declared in a switch list at the head of the block in which they occur. This does not disallow forward jumps; it merely allows the Translator to deduce to what level the jump is to be made. e.g.

```
begin  switch  S1:= FRED, JIM;
          go to JIM;
       FRED:---
       begin switch S2:= JIM;
             ---
          go to JIM;
       JIM:---
          go to FRED;
       end·
       JIM:---
   end
```

This means that there is no necessity to "chain" labels, with all its attendant complexities.

2.2 Expressions that should be of type Boolean may be of type arithmetic. Boolean expressions must reduce to the Boolean constants true and false, which in 903 ALGOL is considered equivalent to the values " ≠ zero " and "zero" respectively. Arithmetic expressions also reduce to these values at run time, and the Translator performs no check on this. As a result, the following constructs are permissible:

```
if a + b then ...
a:= a > b
```

Note: Owing to the stack priorities involved, " a:= a+a>b" is the equivalent of "a:= 2a>b" which will assign the value one or zero to "a" depending on the truth or otherwise of the Boolean expression.

3. THE DATA STORE

The Translator requires the following storage areas;

(1) Name list (NL)
(2) Constants list, which includes label information from switch lists (CODL)
(3) Stack
(4) Buffer Area (INBUF)
(5) Work space area (W)

3.1 The Name List (NL)

The name list contains the names and details of all the identifiers with a current valid declaration. This list is divided into blocks separated by block stoppers.

When a block closes, the Name list is then cleared back to its stopper.

A Name list entry is four words long and contains;

WORDS 1 and 2

(i)	NAME	First six characters not separators. Shorter names are stored left justified.

WORD 3 (from most significant)

(ii)	FML	Set if formal parameter (1 bit)
(iii)	V	Set if call by value Also set during procedure body to throw out recursive call (1 bit)
(iv)	U	If identifier has been used (1 bit)
(v)	Special	This procedure is interpreter not library (1 bit)
(vi)	OWNCOD	Set if procedure is owncode (1 bit)
(vii)	ADDRES	Address of identifier, or if formal parameter the parameter number (13 bits)

WORD 4 (from most significant).

(viii)	FD	Set 1 on procedure assignment Set 3 on leaving procedure body, and can fail trying to assign from outside (2 bits)
(ix)	TYPE	Type of identifier (12 bits)
(x)	DIM	Dimensions of array or switch or number of parameters of a procedure (4 bits)

A Block Stopper contains - 1 in word 1 and BN in Word 2.

3.2 Constants List (CODL)

This list holds the constants used in the source program ; to prevent every constant taking up space each time it is used (as it would if the constants were inserted as they occurred into the object program) CODL is searched as each constant occurs to avoid duplication.

The list also contains details of all switches and label declared during the source program. An example best illustrates its use:

(assume in block 54)

Switch S := LAB1, LAB2, LAB3;

sets up NL and CODL as follows:-

Name List		CODL
name	address (in CODL)	
S	1	+3 (count of labels)
LAB1	2	+0
		54
LAB2	4	+0
		54
LAB3	6	+0
		54

When a label is met preceding a colon, it is looked up in NL; from there the address in CODL is available, and the current program address is entered in CODL in place of the "+0". The block number must be in CODL to discover at run time how many entries should be unstacked on performing a jump to a label.

These addresses have base address added at load time so are distinguishable to avoid their being used as constants at translate time.

Name List Entry

			N		A	M
			E		Space	Space
FML	V	U	special	OWNCOD	ADDRES (13 bits)	
F D			TYPE (12 bits)			DIM (4 bits)

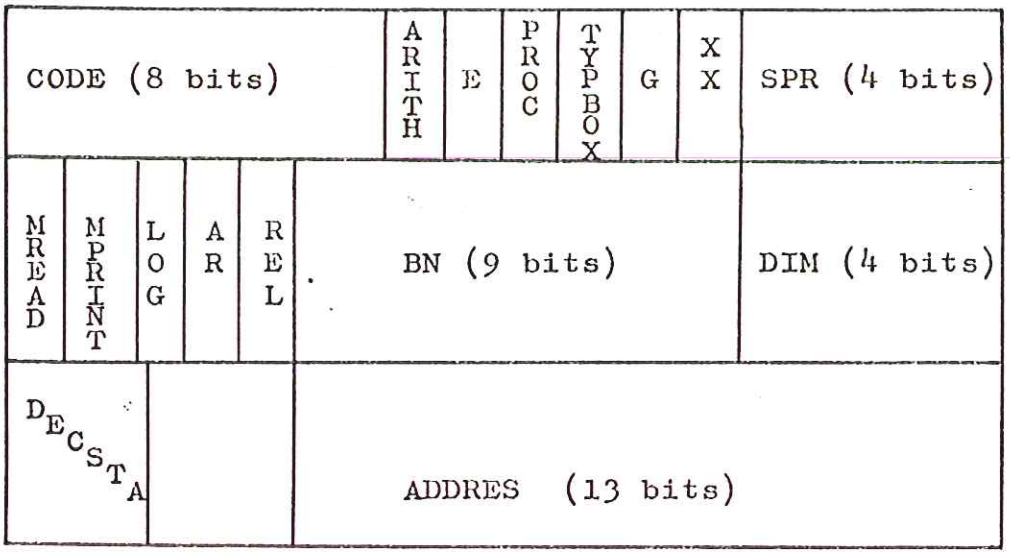
PARAMETER ENTRY

ZERO	V	TYPE (12 bits)	

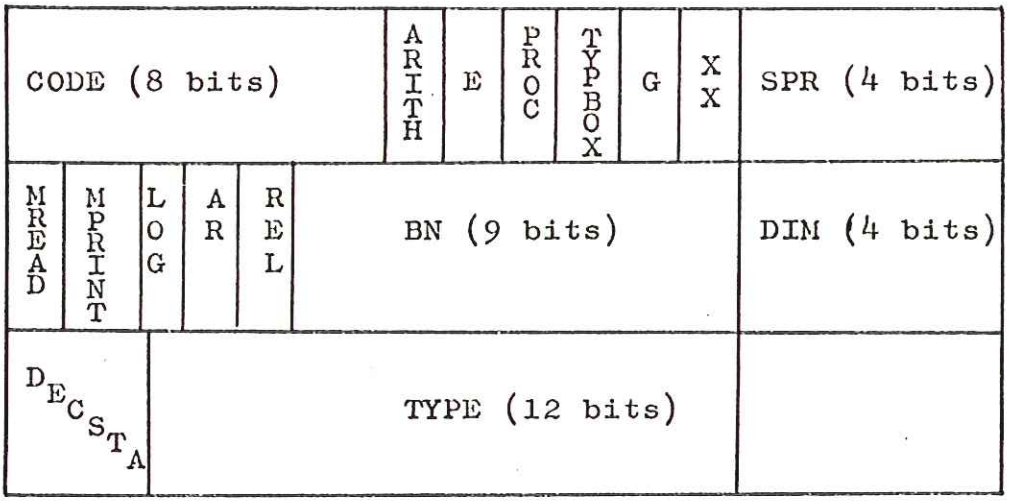
3.3 The Stack

The stack is used as a holding store to enable expressions to be converted intoReverse Polish, and also to deal with the nested statement structure. The next page shows a table of the stack - and compare - priorities used for delimiters. In general, operands are compiled, and operators are stacked. Unstacking is controlled by the stack priority $\bar{}$, and loops until a stack priority is met that is less than that with which the unstacking procedure is called.

Stack Entry 1



Stack Entry 2



3.4 Buffer Area

This area is 40 words long and stores source lines, the filling of this area is automatic. Once the Translator asks for a character either the next is supplied or it is found to be an nlcr and the buffer is refilled up to the next nlcr or a stopcode.

TRANSLATOR STACK

Stacked Item	Stack Priority	Compare Priority	Remarks
[$[_{AD}$	0	void	unstacked by]
(	0	void	unstacked by)
] ; end ,	void	1	not stacked; used only to unstack items
)	void	2	
begin	0	void	unstacked by end
proc begin	0	void	unstacked by ;
for	0	void	unstacked by do or ,
simple	0	void	unstacked by step, until or while
step, until, while	0	void	unstacked by do cr ,
MAMPS	0	void	replaced by $[_{AD}$
if	0	void	unstacked by then
then E	0	2	conditional expression
else E	2	2	
then S	1	2	conditional statement
else S	1	2	
GT, GTF	2	void	go to label) unstacked at
GTS, GTFS	2	void	go to switch) statement end
:=	2	12	
≡	3	3	
⊃	4	4	
∨	5	5	
∧	6	6	
¬	7	7	
> ≥ = ≤ < ≠	8	8	
+ -	9	9	
x / NEG	10	10	
↑	11	11	
IND	12	12	array subscript

3.5 The work space area is 15 words long. Although the coding in referring to the 5th word would address it as W+4, to avoid ambiguity it is referred as W4 and W+4 reserved for W plus the Value 4.

4. THE CHARACTER SET

There are 63 characters in the internal set. Their octal representation is shown below alongside the 503 flexowriter symbol. The 4100 Westrex character is shown alongside in brackets when different.

	00	10	20	30	40	50	60	70
0	Space	(	0	8	≠(\)	H	P	X
1	nlcr	)	1	9	A	I	Q	Y
2	~(")	*	2	:	B	J	R	Z
3	≠(½)	+	3	;	C	K	S	[
4	$(£)	,	4	<	D	L	T	£
5	%	-	5	=	E	M	U	]
6	&	.	6	>	F	N	V	↑
7	≠(/)	/	7	¹⁰	G	O	W	?(←)

e.g. The double character ≠ in the 503 code corresponds to ½ in the 4100 code and is represented internally as 3.

Note nlcr is represented in the westrex code by carriage return, line feed and run out. This is handled on input to make line feed the operative symbol.

Stop code is treated in most respects as an nlcr. It is recognised in GETCHA (FILBUF) and terminates the buffer filling. When the line has been processed the increment line count is omitted and the program pauses.

Becomes (:=) is left stored as two characters but the routine GETCHA has a look ahead facility to cope with this as it does with parameter comments in procedures.

The colon equals sign must not contain a separator.

5. PROGRAM STRUCTURE

A compound statement consists of a set of statements preceded by begin and followed by end. A block, however, has one or more declarations between begin and the set of statements which are again followed by the delimiter end. On meeting the first declaration after a begin, therefore, a block is implied.

Whether or not an object block is set up depends entirely on the mode of storage used for the variables local to that block. The "wipe-off" mechanism at the end of an object block is best achieved by clearing back the stack pointer, without worrying whether to "block off" other areas of storage. There are two extremes to this problem:

i) that all variables are kept on the stack, and hence there must be an object block generated for every source block; or

ii) that there is so much room in store that all variables are given an absolute address in store and no object block need ever be generated. (Remember that we do not have to worry about recursion).

This translator steers a middle course; all scalars are given addresses in store, and arrays, formal parameters etc., are kept on the stack. As a result, an object block is only set up for the latter cases. This approach should be contrasted with Randell & Russell's method, which keeps all variables on the stack.

For object program operations see the pord manual.

6. TYPE HANDLING

6.1 General

The conversion of variables and expressions from real to integer or from integer to real is handled by the translator. Basically, the rules for determining the type of an expression are as follows:-

(i) The type of an arithmetic operation is integer if both of the operands are of type integer, otherwise real.

* e.g. (a + b)x D

a + b gives type integer; since D is real, (a + b) is converted to real before the multiplication is performed.

(ii) In an assignment statement, the expression on the right-hand side is converted to the type of left part list.

e.g. r := i := q + T

q + T gives type real since T is real, and is converted to integer for the assignment to i. r being integer required that i, and any more elements of a multiple assignment should be of the same type.

There are three conversion operations in the object code:-

(i) R to I convert 2nd operand to integer
(ii) I to R1 " " operand to real
(iii) I to R2 " 1st operand to real

e.g. a:= b + (D - e)

The object code in reverse polish is:-

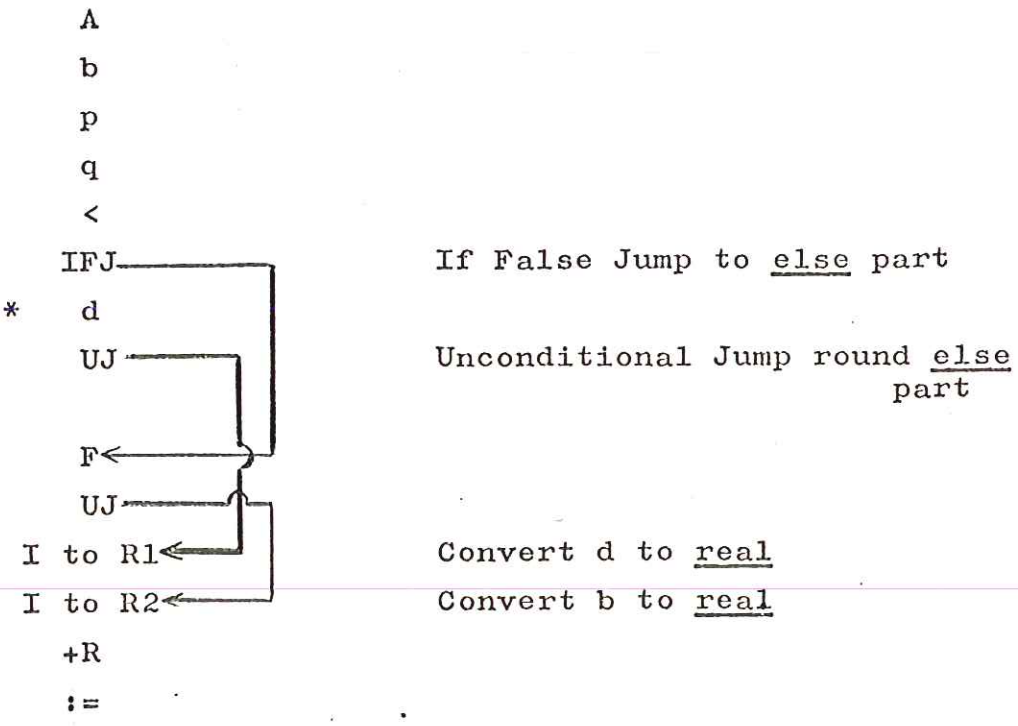
```
a
b
D
e
I to R1        convert e to real
- R
I to R2        convert b to real
+ R
R to I         convert b + (D-e)
:=               to integer
```

The translation of conditional expressions when type conversion is involved, is illustrated in the following example:-

```
A := b + (if p < q then d else F);
```

* a - z denote a variable of type integer
A - Z " " " " " real

Since the else part produces a real result, the then part must also produce a real result. The object code in reverse polish is:-



* The conversion (I to R1) cannot be placed here since in a single scan the translator is unable to know that the else part will give a real result.

6.2 Rules for Type Determination

(i) Assignment Right hand part converted to left-hand part.

a := b + D; (b + D) is converted to integer

(ii) Arithmetic or relational operation If either operand is real,

a := (b + D) x e; b and e are converted to real.

a := if B < d then p + q else y - R;

d is converted to real since B is real (p + q) and y are converted to real since R is real.

(iii) Subscript expression The subscript expression must give result integer.

a := A [b + D] ; b is converted to real since D is real.

(b + D) is converted to integer since it is a subscript.

(iv) For Statements The list elements must be converted to the same type as the controlled variable.

```
for V := a,b step d until e, f while
p < q do .....
```

a,b,d,e and f are converted to real since V is real.

(v) Actual Parameters In a procedure call, the actual parameters must be converted to the type (real or integer) of the corresponding formal parameters.

procedure P (a,B); value a,B; integer a; real B;

P (A,b); A is converted to integer and b to real to match the types of the formal parameters a and B.

(vi) Division Except when specifically required through use of a special Algol word div, the result of a division is real irrespective of the types of the operands, though there are 2 division operations in the object code. This is best illustrated in the following table:-

ALGOL expression	Conversion	Object code operation	Result
a/b	-	/ integer	real
A/B	-	/ real	real
a/B	I to R2	/ real	real
A/b	I to R1	/ real	real
a div b	(fail if either real)	div	integer

(vii) Exponentiation This is similar to division, but there are third and fourth object codes for exponentiation.

(A) Integer exponentiation giving result integer. This operation may only occur when the mantissa is integer and the exponent is a positive integer constant.

(B) A special primitive for R↑i stops the expression being failed out when the integer is negative.

The operations are illustrated below:-

ALGOL expression	Conversion	Object code operation	Result
a↑b	-	↑integer (1)	real
A↑B	-	↑real }	real
a↑B	I to R2	↑real } (2)	real
a↑integer constant	-	↑integer (3)	integer
A↑b		↑(special(4)	real

6.3 Type determination in the Translator

A global variable TYPBOX contains the current type (real or integer) of an ALGOL arithmetic expression. This is set by every arithmetic identifier or constant when compiling the object code for the operand (in TAKE) and is stacked with the binary operator which follows. When the expression is unstacked this type is unstacked into LOKTYP, TYPBOX is set to the resulting type of the expression and any conversions necessary are compiled.

Example

A := d + E*f ;

Translator Stack	TYPBOX	Object Code		
:=(R)	real	A	A :=	
+(i)	integer	d	d +	
*(R)	real	E	E *	
	integer	f	f ;	unstacks the statement
	real	I to R1 *R		convert f to real
	real	I to R2 +R		convert d to real
	real	:=		

On unstacking the decision of whether to compile a conversion or not, is made by comparing TYPBOX with the type stacked with the operator.

TRANSLATOR Conversion Rules

(i) Each identifier or constant sets TYPBOX to its type (real or integer) in TAKE.

(ii) The type from TYPBOX is stacked with + - * / , < < = > > ≠, :=, [, else, for comma, for:=, step, until, while. With the exception of [, the type stacked is that of the preceding variable or expression. In the case of [it is the type of the preceding array identifier. Boolean is stacked as integer.

(iii) In the subroutine UNSTAK, the type stacked with the operator (e.g. + i) is compared with the current type in TYPBOX. A conversion is compiled if necessary and the relevant operator compiled (e.g. + R). The current expression type is then placed in TYPBOX.

(iv) Conditional Expressions The type of the then part is stacked with else. On unstacking the else part, the type stacked with else is compared with the type of the else part (in TYPBOX) and a conversion compiled where necessary.

(v) The operators ↑ and / always leave TYPBOX set to real on unstacking except for div and I↑I as noted in 6.2 (vi) and (vii).

(vi) The assignment operator := requires a conversion to the type stacked with it.

(vii) Array element On unstacking a subscript expression [requires the type to be integer and a conversion is compiled if the expression is real. The type of the array, stacked with [, is then placed in TYPBOX.

(viii) Procedure call If the call is to a type procedure (i.e. it yields a value) the type of the procedure is set in TYPBOX after compiling the call at) or TAKID.

(ix) The expression bracket '(', if and then do not require a type to be stacked with them nor do they change TYPBOX on unstacking.

(x) For statements The type of the controlled variable is stacked with the start of each list element. On unstacking the list element expression, a conversion to the stacked type is compiled if necessary, and the type again stacked with the start of the next list element.

(xi) Actual Parameters These are dealt with in the subroutine PRAMCH which determines whether to compile a conversion or not by comparing type of the actual parameter with the type of the formal parameter (this information is found in the Namelist). TYPBOX holds the actual parameter type, if this is an expression.

COMPARISON between operator type and TYPBOX

Operand 1 See Note	Operand 2 See Note	Object code conversion expression	else
integer	integer	op I	-
real	real	op R	-
real	integer	I to R1, op R	I to R1
integer	real	I to R2, op R	UJ, I to R1

Operand 1 See Note	Operand 2 See Note	:=	/	[
integer	integer	:=	op I	-
real	real	:=	op R	R to I
real	integer	I to R1, :=	I to R1 op R (Special R I)	-
integer	real	R to I, :=	I to R2 op R	R to I

Note op R, op I denote an operator of type real or integer. Operand 1 is the variable or expression preceding the stacked operator, or the left part of assignment, or the then part and conditional expression, or the array identifier.

Operand 2 is the variable or expression following the stacked operand, or the RH side of an assignment, or the else part in a conditional expression or the array subscript.

7. NOTATION

The action of taking the item at the top of the stack and distributing the various constituent parts of the item into fixed locations is denoted by the procedure RESTO. The parameters to this procedure correspond to some or all of the constituent parts of the item at the top of the stack. Those parts that are to be stored in fixed locations are indicated by a parameter, enclosed in square brackets, giving the name of the location. The final action of RESTO is to decrease SP (the stack pointer) by one, and setting TS to be the current top of stack.

For example, if the item at the top of the stack is

begin TR , 53, 1026, 0

then RESTO [..... BN , Q]

deletes this item, having set BN to be 53 and Q to be 1026.

The procedure PRESTO is a variant of RESTO which does not decrease SP and TS then remains the same.

The subroutine STACK has, as parameters given on separate lines and enclosed in square brackets, any items which are to be added to the stack. The stack priorities are indicated by the final underlined integer.

For example,

STACK [DECSTAT, NLP, BN, PP
proc begin 0]

will stack the item "DECSTAT, NLP, BN, PP" and then the item "proc begin , 0".

The subroutine COMPIL uses a similar notation to indicate any operations (and their parameters) to be added to the object program.

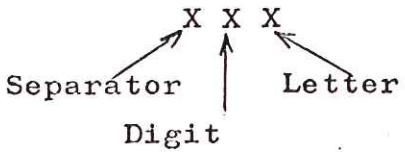
A subscripted item has the subscript value enclosed in square brackets. If an identifier is enclosed in round brackets this may be taken as 'contents of'. e.g.

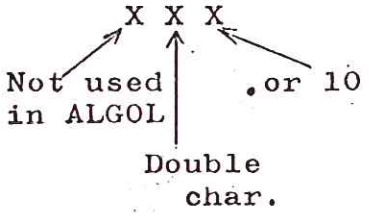
$$(NLP) := \overset{-1}{BN}$$

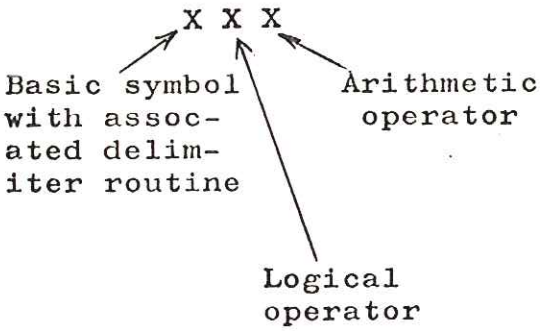
is putting a block stopper in the name list. -1 is put where the name list pointer points and the Block Number goes in the next word, Replacing the Global Block Number; BN := (NLP+1) rather than BN := NLP[1].

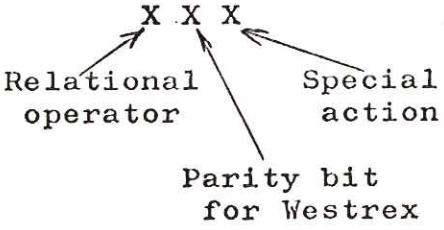
To make stacking easier, the Global variables take the values corresponding to the stacked position. E for example is stacked as the 9th significant bit of word one, and consequently the variable is carried about as 0 or +256 in the coding. The flowchart however, refers to E as 0 or 1. It is thought safer for other multistate variables to refer to their actual values (DECSTA := /0 0).

Values of masks are not shown in the flowcharts: GRPCOD := masked TABLE [LASTCH+1] implies that the mask is the 12 most significant bits, and in agreement with SIR conventions the bits mean:









Also, similar to SIR, there exists a variable called OPTION which varies the action of the translator.

The bits mean:

1	Halt on error
2	Warning mode
4	Output check functions
8	Inhibit library scan

The method of testing is shown as e.g.

OPTION
2 bit in a decision box.

Finally, in dealing with the flowcharts, certain abbreviations have been used, but it is hoped that in the main they are self evident, some examples are given below.

nlcr	New line carriage return
inc	Increment (Var := Var + 1)
Acc	Accumulator
Aux	Auxiliary register
Str	String
Real/int/Bool) R / I / B Scalar)	real, integer or boolean

Note The symbols $\wedge$ and $\vee$ are also used in logical tests e.g.

$$(E=0) \wedge (M=1)$$

8. Reference Lists

It has been found convenient to have certain lists available for immediate reference. These are:

(i) Types (bit patterns corresponding to types of variables);

(ii) Delimiter 8 bit values (in practice stored at most significant end);

(iii) Correspondence of routines to error numbers;

(iv) Glossary of Global variables used.

(i) Types

a	b	c	A	B	C	D	E	F	G	H	J		
1	0	0	0	1	1	0	0	0	1	0	0	real	106100
1	0	0	0	1	1	0	1	0	1	0	0	integer	106500
1	0	0	0	1	0	1	0	0	1	0	0	boolean	105100
0	1	0	0	1	1	0	0	0	1	0	0	real array	046100
0	1	0	0	1	1	0	1	0	1	0	0	integer array	046500
0	1	0	0	1	0	1	0	0	1	0	0	boolean array	045100
0	0	1	1	1	1	0	0	0	1	0	0	real procedure	036100
0	0	1	1	1	1	0	1	0	1	0	0	integer - procedure	036500
0	0	1	1	1	0	1	0	0	1	0	0	boolean - procedure	035100
0	0	1	0	0	0	0	0	0	1	0	0	procedure	020100
1	0	0	1	1	1	0	0	0	1	0	1	real procedure zero	116120
1	0	0	1	1	1	0	1	0	1	0	1	integer - procedure zero	116520
1	0	0	1	1	0	1	0	0	1	0	1	boolean - procedure zero	115120
1	0	0	0	0	0	0	0	0	1	0	1	procedure zero	100120
0	1	0	0	0	0	0	0	1	0	0	0	switch	040200
1	0	0	0	0	0	0	0	1	0	0	0	label	100200
0	0	0	0	0	0	0	0	0	0	1	0	string	000040

a must not be followed by bracket
b must be followed by [bracket
c must be followed by (bracket

A type procedure
B Algebric (Arith $\vee$ boolean)
C Arithmetic

D Boolean result
E Integer result
F Switch or label

G not a switch, label or string
H String
J Some procedure zero (parameterless)

(ii) Delimiter 8 bit Codes (internal entities)

Octal	Decimal	
	0 - 63	As internal code for letters etc.
100	64	Spare
	65	go to
	66	if
	67	for
104	68	end
	69	print
	70	read
	71	begin
110	72	code
	73	algol
	74	comment
	75	boolean
114	76	integer
	77	real
	78	array
	79	switch
120	80	procedure
	81	string
	82	label
	83	value
124	84	true
	85	false
	86	≤
	87	≥
130	88	≠
	89	≡
	90	implies
	91	or
134	92	and
	93	not
	94	then
	95	else
140	96	do
	97	:=
	98	step
	99	until
144	100	while
	101	div
	102-127	spare
200	128	spare
	129	then E
	130	then S
	131	begin TR
204	132	begin ALL
	133	for begin
	134	simple
	135	else E
210	136	else S
	137	GTF
	138	GT
	139	$\lceil_{AD}$
214	140	IND
	141	proc begin
	142	GTS
	143	GTFS
	144	STA
	145	NEG
	146	MAMPS

(iii) Table of error numbers

1	OUT (Read)
2	OUT (Print)
3	SETPRO
4	SWITCH
5	PRAMCH ACTOP RRBRAK QUOTE
6	ACTOP PROCED
7	NUMBER BECOMS
8	NUMBER
9	COLON
10	BCR
11	BCR
12	BCR
13	BCR
14	
15	EVALNA
16	FCLAPS
17	SEARCH PROCED
18	SEARCH
19	OUT
20	ENDSTA
21	FOR STEP
22	TAKID
23	RSBRAK
24	LSBRAK
25	LRBRAK
26	SWITCH
27	DECL
28	BECOMS
29	COLON
30	TAKE AOP
31	TAKE TAKID
32	ENDSTA
33	LSBRAK BCR
34	UNSTAK
35	EXP COLON QUOTE
36	DEC
37	
38	SEARCH
39	RSBRAK
40	END
41	TAKID LRBRAK
42	GOTO
43	FORCOM
44	FOR
45	TAKE

```
46      TAKID
47      REAL
48      SEARCH    COLON
49      ACTOP
50      ARRBND    TITLE

51      PRAMCH    RSBRAK    RRBRAK
52      BECOMS
53      SEMICO
54      DEC
55      EXP

56      TAKCHA
57      AOP
58      RLT    LOGOP
59      LOGOP
60      BEGIN

61      SEARCH    LRBRAK
62      LRBRAK
63      DEC
64      UNSTAK
65      PROCED

66      ARRBND
67      IF
68      IF
69      THEN
70      ELSE

71
72      ARRAY
73      LSBRAK
74      RSBRAK
75      RSBRAK

76      REAL
77      COMMA
78      STEP
79      NCLAPS
80      STEP

81      RRBRAK
82      LRBRAK    RRBRAK
83      STACK
84      RRBRAK
85      CHECK    PROCED

86      PROCED
87      SEARCH
88      PROCED
89
90      PROCED

91      GETCHA
92      PROCED
```

```
 93     STATRM
 94     PRAMCH    PROCED
 95     RSBRAK

 96     FORCOM
 97     THEN
 98     GETCHA
 99     SEARCH    FOMCOM
100     IF

101     PROCED
102     PROCED
103     ARRBND
104     UNSTAK
105     QUOTE

106     COLON
107     FOMPIL
108     PRAMCH
109     PROCED
110     PROCED

111     RRBRAK
112     BECOMS
```

(iv) GLOBAL VARIABLES

Variable	Use	Set in
ADDI	local copy of ADDRES (I) ADDI+1 DIM (I) ADDI+2 FML (I) ADDI+3 Used (I) ADDI+4 V (I)	FOMCOM) ADJI
ADDRES	Temporary storage for unstacked address	UNSTAK
AR	Not now used	
ARITH	Checks Validity of use of rel., or log. operators (fail if ARITH = 1)	SET to 1 in [, STEP SWITCH Cleared in ENDSTA DO ElseS if)] ARRAY] DECL
ARRCOU	Array count on array declaration	
BN	Current block Name BN+2 Last found block Name BN+4 Highest block Name used	DO FOR DEC) PROCED NCLAPS) START UNSTAK SEARCH) DEC FOR PROCED)
BUFLAG	Marker to show when input buffer exhausted	START GETCHA
C	Zero constant	START
CHKSUM	Binary sum of characters for checking binary output	PUNGRP
CNL	Marker set during Name list collapse	NCLAPS
CODE	Used by PUNGRP to inform the loader of type of output	COMPIL FOMPIL PUNGLB
CODLP	Relative pointer to next free in CODL	START SECODL
CONS	0; real constant. 1; integer, 2; Boolean (true/false)	NUMBER BCR

Variable	Use	Set in
CONSTA	Current constant in ALGOL text	NUMBER BCR
DECSTA	0 at begin, 1 during declarations 2 during statements.	DEC FOR PROCED BEGIN GOTO IF BECOMS COLON (ARRAY REAL PROCED SWITCH]
DECTYP	Type of current declaration	BCR QUOTE AOP BCR
DELIM	Current delimiter DELIM+1contains the associated GRPCOD	UNSTAK
DIM	Temporary storage for unstacked dimensions	
E	Set 1 for statement, 0 for expressions	Set to 1 at ENDSTA ACTOP SERPRO ELSE DO] (Set to 0 at EXP PRINT QUOTE IF GOTO [FOR
EXPRES	Set for expression bracket in print list	SETPRO (
EXPTYP	Records expected type(s)	Set to Switch/Label in GOTO Cleared at ENDSTA ELSE IF ,) [
F	For clause marker	DO FOR GOTO
FALCOU	Register to accumulate error number	FAIL
G	Marker for syntax checks (for clauses)	FOR STEP WHILE UNTIL
GRPCOD	Information code for current delimiter, or character (Sec. 7)	TAKCHA GETCHA
I	Contains found Value of NLP	SEARCH]
LASTCH	Last character read from buffer LASTCH+1 Present character LASTCH+2 Next character	)) GETCHA)) EVALNA)

Variable	Use	Set in
LASTCP	Last item compiled	COMPIL
LASTDL	Last delimiter LASTDL+1 GRPCOD of LASTDL	BCR
LEVEL	Not now used	
LHTYPE	Left hand type of an assignment	BECOMS
LINE	Number of line being currently processed	FILBUF
LOG	Not now used	
LOKTYP	Space reserved for unstack in TYPBOX	UNSTAK STEP
M	0 if delimiter only; 1 if identifier delimiter; 2 if constant delimiter	IDENT NUMBER END BCR
MFAIL	Records a failure	FAIL
MPRINT	Set in print statement	PRINT LSBRAK LRBRAK ENDSTA
MREAD	Set in Read statement	READ LSBRAK LRBRAK ENDSTA
NAM	Contains Current (or last) identifier	IDENT
NDAP	Notional data area pointer	START
NLP	Name list pointer	START NCLAPS SEARCH DEC PROCED FCLAPS
	NLP+1 Initial Value of NLP (7995) NLP+2 Value of NLP after 1st Block	NCLAPS
OPTION	Parameter of Translation	START
OWNCOD	Set for owncode procedures	PROCED CODE
P	Parameter for checking validy of some subroutine calls.	BCR EXP TAKE DEC
PARAM	Now now used	
PH	Set when processing a procedure heading	PROCED
PP	Current value of pord pointer	COMPIL
PRCENT	Temporary storage for found entry in Name list	PROCED

Variable	Use	Set in
PROC	Set for procedures	PROCED LSBRAK LRBRAK QUOTE
PROCPO	Pointer to procedure name in actual call	LRBRAK
PRMCOU	Number of current parameter	PROCED RRBRAK
Q	Used to search constant area	SECODL
R	Not now used	
REL	Not now used	
SP	Stack pointer	RESTO STACK
SPR	Stack priority.	UNSTAK
SV	Variable to denote where an array element is syntactically value	ENDSTA GOTO BECOMS
TS	Top of stack	RESTO STACK
TYPBOX	Type of current Algol section	SEARCH UNSTAK LSBRAK TAKID TAKE TYPCHK
TYPE	Not now used	
WANTED	Marker to show whether a library function is required	FOMPIL
WM	Warning message marker	FCLAPS NCLAPS END
XX	Used for checking validity of : and , in array declarations	ARRBND

9. Notes on Translator listing

(i) Location of entities

An entity can be located in the listing by bearing in mind the Group to which it belongs. There are four groups -

Vol 1 has Group 1	All global variables Code conversion table Basic routines like PRINT and GETCHA
Vol 2 has Group 2	Routines such as COMPIL which use Group 1 routines
Vol 2 has Group 3	Routines such as DECL which use Group 2 routines.
Vol 3 has Group 4	A routine for each delimiter e.g. array SEMICO etc.

Within each group the order is alphabetical as far as possible. At the end of the whole translator is the Central Loop in order that jumps to delimiter routines can be to located places rather than forward jumps. This ordering helps to shorten the relocatable binary tape.

(ii) Assembly

1. Assemble using 6th April 1966 SIR to a relocatable binary tape.
2. Input this tape which produces a global label list, and also records the store used.
3. Clear locations 6000-8178 so as to remove the loader.
4. Input the initial names list in binary which occupies from 7800 approximately to 7998
5. Dump the store using "Larry T22".
6. Attach a clear stores (or punch it in) at the front.

(iii) Initial Names list

The initial names list must be assembled using a version of SIR in 2048. After assembly clear locations 8-7168 to remove SIR and then dump the store using Larry T22.

CENTRAL LOOP

The basic cycle routine (BCR) fetches the next ALGOL "section", which is in one of the following three formats, and sets the variable M to be

0 : delimiter by itself
1 : identifier followed by delimiter
2 : constant followed by delimiter

A delimiter being a basic symbol like ; or +, or an ALGOL word like begin.

Depending upon the delimiter discovered, a transfer of control is made to the relevant routine, which will end with a jump back to OUT. In some cases BCR is called during the routines (e.g. in procedure it is called to check whether the procedure body is in ALGOL or in own code) and in these cases exit is made back to OUT2, and processing continues on that delimiter.

BCR is called with one parameter which can be either 0, 1, 2, 3 or 4. The first three require the section to comply with this forecast for M. BCR (3) is the general call which will accept any section, and BCR (4) is the call during subsequent discarding of the rest of an erroneous statement or declaration, or of a comment.

ERRORS

FAIL 33) or] precedes constant or identifier

FAIL 10 M doesn't agree with the parameter in BCR (0, 1 or 2)

FAIL 11 Alphanumeric character, '.' or '$_{10}$' misplaced.

FAIL 12 Constant or identifier true false

FAIL 13 comment misplaced

FAIL 15 Unrecognised underlined word.

FAIL

This routine is accessed whenever a failure occurs which doesn't make continuation impossible, N.B. If the stack Pointer moves beyond the beginning of the stack this is considered impossible and the program jumps straight to ENDPRO. In general, the action taken is to throw away the remainder of the statement.

First the error message is printed which gives the error number, line number, line stored in INBUF and a pointer which shows which is the offending character (except in the case of an illegal character when the whole buffer is inspected and illegal characters replaced by ? or ← (westrex). The error count is incremented and the program terminates if it reaches 20. The option is inspected and set to checking mode, followed by a pause in the 'halt on error' condition. It is here that a return is then made to NCLAPS if an unallocated label has been found, otherwise the stack is cleared back to a begin, and the global variables are reset.

The current delimiter is then inspected. If not a ;, end or begin, the next ALGOL section is brought in to replace it, and this is repeated until one of these delimiters occurs. FAIL then exits to either the routine BEGIN, or internally to END or SEMICO unless the text is found to be inside a procedure Heading when the rest of the heading is discarded before processing.

TAKCHA (Take Character)

This subroutine, called from BCR, places the next significant character in the source line into the global variable space. The current character is moved to LASTCH, the character just received is put to LASTCH+2 for the 'look ahead' facility, while the character currently there becomes the present character to be processed, and moves to LASTCH+1, "space" or "tab" are ignored; "newline" replenishes the source line in the buffer by the use of the subroutine FILBUF.

The only characters requiring further study are : and). The former may be followed by = to produce :=, and the tests on this branch are used to ascertain whether this is the case.

The latter, closing round bracket, may be the start of a comment acting as an actual parameter delimiter.

E.G:

```
PROCCALL (a) this is a comment :(b);
is the equivalent of PROCCALL (a,b);
```

PROC is tested to see whether we are dealing with a procedure call, and if so, the next character in the line is tested to see if it is a letter. This differentiates between

```
PROCCALL (a) this is
and PROCCALL (a);
```

If this is a comment, the buffer is searched for the terminating colon ; when found, the next character is checked to be (, and THISCH is set to comma (the other valid form of parameter delimiter).

ERRORS:

FAIL 56 ; character other than separator between the : and (of a parameter comment.

IDENTIFIER

This routine merely builds up the identifier character by character in the state variable NAM, ignoring any significant character after the sixth (an IFIP ALGOL subset restriction).

It is called from BCR when the next character is found to be alphabetic, and reads characters storing them in the most significant end of the double length pair NAM and NAM +1. It sets M to 1 and exits when a delimiter is found.

EVALNA

This subroutine, called from BCR, converts a delimiter such as ~ begin~ into an integer. The routine clears a location then adds successive characters to it multiplying each intermediate result by 67. When the closing character is met, the result is looked up in a table and replaced by the required delimiter .

ERROR

FAIL 15; Unrecognised result after this evaluation.

NUMBER

This subroutine (written by C.W. Nott of N.P.L) ends by placing the binary equivalent of the decimal digits in the state variable CONSTA and CONSTA+1.

This routine calls its own local routines; STAND which standardises the floating point number in W3 and W4 (binary exponent in W5); POWER which multiplies the number by the accumulator to the power of 10. Bearing in mind also the uses of the variable space (below) the routine should be clear from the flowchart.

LASTCH	last character read.
Sign	set to one if the exponent part is negative.
W3,4&5	used for the partially computed number.
Exp	set to one by the character '$_{10}$'
Point	set to one by the character '.'
Dec	used to count the number of decimal places and for any exponent.
Max	set to one if the integer or exponent part of the number exceeds capacity.
PWS	parameter for POWER
PMKR	marker to remember whether this power was negative or not.

ERRORS

FAIL 7; illegal number

FAIL 8; integer number too big.

BASIC CYCLE ROUTINE (BCR)

This subroutine is used to fetch the next section of ALGOL text, and to allow for comments after the delimiters ; and begin. It is controlled by its parameter P as follows:

(1) P = 0,1, or 2 : a certain type of ALGOL section is expected, and this is checked by comparing the final value of the state variable M ;

(ii) P = 3 : no check is made on the type of the ALGOL section; and

(iii) P = 4 ; the subroutine is merely being used to find the next delimiter, such as in an end comment, or during scanning after an error (see FAIL). The constituents of identifiers and numbers need not therefore be processed.

BCR itself uses several subroutines. First it calls TAKCHA, which brings up the next character which is tested to distinguish between delimiters, numbers and identifiers.

A ~ (or " in 903 ALGOL) announces an ALGOL word (see Notes on Internal Character Set), such as ~ b e g i n ~ . This is operated upon by the subroutine EVALNA to convert the string of characters into an integer, and the delimiter list is then searched. If the delimiter is comment, further characters are taken until ; is met, which ends the comment. If the delimiter is true or false CONS, M, and CONSTA are set. Otherwise this joins the basic symbol path.

If this character is a basic sysmbol, the symbol list is searched, P is checked where necessary and the routine finishes.

If this character is a digit, decimal point or 10, NUMBER is called to place this number in the variable CONSTA.

Similarly in the case of a letter, IDENT is called to place this identifier in the variable NAM.

ERRORS

FAIL 13; comment does not follow; or begin.

FAIL 11; letter, digit, '.' or '10' misused.

FAIL 10; identifier or constant not as expected.

FAIL 12; true or false preceded by constant or identifier.

FAIL 33;] or) precedes constant or identifier.

FAIL 15; unrecognised ALGOL word (EVALNA).

UNSTAK

This subroutine is used to unstack items from the top of the stack until an item is reached whose stack priority SPR, is less than the value given by the parameter, or until the stack is empty. In general, items are unstacked directly into the object program (in some cases after extensive typechecking) with the following exceptions:

(i) If the stack priority SPR of the current item is 12, the variable P (set up by TAKE or EXP) determines whether to generate INDA or INDR.

(ii) If the SPR of the current item is 8 (i.e. it is a relational operator) the parameter should not be 8 as this would mean an incorrect use of relational operators, e.g:

$$x < y + z < 4$$

(iii) If the top of the stack is else E, TYPBOX and LOKTYP must be checked to produce the special case;

```
UJ  PP+2
Update ADDRES
I —> R1
```

when LOKTYP is zero and the else part turns out to be real. (This is a consequence of the single pass technique).

(iv) Another special case is where the delimiter from the stack is :=, the necessary conversion may be R→I if the left hand side is integer.

(v) Where the delimiter from the stack is an arithmetic or relational operator, further typechecking is necessary, and special primitives may be compiled, as in the cases of integer ↑ 2 or Real ↑ integer.

ERRORS

FAIL 34; incorrect use of relational operator.

FAIL 64; illegal use of subscript variable.

FAIL 104; Div has real operand.

EXP

This subroutine is used to check the validity of use of the current delimiter and to change the state variable E to expression level if necessary.

The parameter P has the following meaning:

(i) P = 1 Delimiter can only be used at expression level.

(ii) P = 2 Delimiter can only be used at statement level.

(iii) P = 3 Delimiter can be used at either level.

UNSTAK is called with a parameter of 12 to unstack IND (the parameter P deciding whether INDA or INDR is to be generated), and the top of the stack tested to see whether := necessitates a changing of E to zero. Logical, arithmetic and relational operators in statements other than procedure calls are failed here.

ERRORS:

FAIL 35; illegal statement - delimiter misused.

FAIL 55; go to, : or for used in expression.

PRAMCH

This subroutine is called from ACTOP to check the actual parameter with the correcponding formal parameter and to compile the relevant object program operation which will access this actual parameter for the procedure that is being called.

TABLE 1 gives a list of formal parameter types with the possible actual paramete types. In general, the rules are :-

(i) If the F.p. is called by name, the a.p. must have the same type and kind.

(ii) If the f.p. is a scalar called by value, the actual parameter may be a scalar, expression, array subscript, constant or type procedure call.

(iii) If the f.p. is a label called by value, the actual parameter may be a label or switch subscript.

(iv) Procedures, strings and switches and parameters of formal procedures may be called by name only.

The choice is further complicated, in the case of scalars, by the fact that an actual parameter may itself be a formal parameter. Table 2 gives the possibilities for this case.

The subroutine uses the following variables:-

PROCPO : namelist address of procedure using this actual parameter.

PRMCOU ; number of actual parameter in the procedure call.

I: namelist address of actual parameter.

TYPBOX : expression type (real or integer)

V : 0 if called by name, 1 if called by value.

The first job of PRAMCH is to recognise whether the parameter is of a formal procedure or not, then W locations 7 to 10 are set up the formal entry, V formal, type of formal and type difference (actual type - formal type). Next in the cases PRAMCH (0) and PRAMCH (3) when parameter checking words are compiled W14 is set with the check word, then W11 is set with a bit pattern according to W7, the formal entry, which later helps with the processing of

identifiers, non-formal function designators and array subscripts.

Little remains now except the somewhat tedious syntatical checking and to compile the parameter call. After this is compiled the parameter checking word if applicable and the routine exits to) or comma.

ERRORS

FAIL 5; fp is not label when ap is a switch subscript.
fp is not called by value when ap is a switch subscript.
fp is not scalar when ap is procedure call or array subscript.
fp is not scalar when ap is expression.
fp is not called by value when ap is expression.
ap is integer constant when fp is non integer by name.
ap is Boolean constant when fp is non Boolean.
ap is real constant when fp is non real by name.
ap is wrong type array when fp is array by value.
illegal ap called by value.
ap is not label when fp is label by value.
ap is not Boolean when fp is Boolean scalar.
wrong type of formal parameter scalar used as ap when fp is called by value.
wrong type of formal parameter used as ap when fp is not scalar.

FAIL 51; too many actual parameters.

FAIL 94; unrecognised formal type.

FAIL 108; parameter of formal procedure called by value.

TABLE 1

F.P.	Actual Parameter	Formal Parameter	Name or Value	Actual Parameter Operation
scalar	Integer identifier/constant	Integer	N	Take address or
	Real " "	Real	N	Take address or
	Boolean " "	Boolean	N	Take constant address
array	Integer array	Integer array	N	
	Real "	Real "	N	Take address of array map
	Boolean "	Boolean "	N	
procedure	Blank procedure	Blank procedure	N	
	Integer "	Integer "	N	Take address of procedure
	Real "	Real "	N	block
	Boolean "	Boolean "	N	
switch	Switch	Switch	N	Take constant area address
string	String	String	N	Take address of actual string
label	Label	Label	N	Take label address
	Label	Label	V	Take label address
	Switch subscript	Label	V	Index label address (INDS)
array	Integer array	Integer array	V	Take address of array map
	Real "	Real "	V	
	Boolean "	Boolean "	V	
scalar	Integer/Real constant	(Integer	V	Take value and compile type
	" " "	(Real	V	conversion if necessary.
	Boolean constant	Boolean	V	Take value.

TABLE 1 continued

F.P.	Actual Parameter	Formal Parameter	Name or Value	Actual Parameter Operation
scalar	expression	(Integer	V)	Compile type conversion if
	"	(Real	V)	necessary.
	"	(Boolean	V)	
	Integer/Real identifier	(Integer	V)	Take value and compile type
	" " "	(Real	V)	conversion if necessary.
	Boolean identifier	Boolean	V	Take value.
	Integer/Real array subscript	(Integer	V)	Index value and compile type
	" " " "	(Real	V)	conversion if necessary.
	Boolean array subscript	Boolean	V	Index value.
	Integer/Real procedure call	(Integer	V)	Compile type conversion
	" " " "	(Real	V)	if necessary.
	Boolean procedure call	Boolean	V	

Actual Parameters which are Formal Parameter Scalars

Example

```
real  procedure  P (t) ;  value t;  integer t;
      begin real R;
       R := t

      end
       .
       .
       .
       .
      procedure Q (a,b,c); value a; integer a,b; real c;
        begin real Y;
          Y := P (a) + P (b) + P (c);
      end
       ;
       .
       .
   Q  (JIM, FRED, BILL) ;
```

The call of procedure Q with actual parameters JIM, FRED and BILL will give value, address, address on the stack since the formal parameters are value a, name b,c. However, the procedure P expects an integer value on the stack, since its formal parameters t is called by value. So the final two parameters are called by Take Formal Value which puts the result in the stack.

In the following table of actual parameter operations, procedure Q would be SOURCE and procedure P would be DESTINATION.

TABLE 2

SOURCE		DESTINATION		Actual Parameter operation
value	I/R	value	I	Take Formal Value. R to I if necessary
value	I/R	value	R	" " " I to R "
value		name		Take Stack Address (R/IFUN) same type
name	I/R	value	I	Take Result Call by Name R to I if necessary
name	I/R	value	R	Take Result Call by Name I to R if necessary
name		name		Take Formal Address same type

SEARCH

This subroutine searches the Name List for the current identifier, and is called with one of four parameters:

(1) w = 0 ; search the entire name list and find the identifier ;

(2) w = 1 ; search the part of the name list local to this block and do not find the identifier;

(3) w = 2 ; search the part of the name list local to this block and find the identifier ; and

(4) w = 3 ; as search (1) except the identifier is then inserted.

In the case of w = 0, each time a block stopper is passed, the block number stored with it is copied into BN+2 (Found Block Number).

When the identifier is found (and that is what is desired), it is marked as used (for later checking in NCLAPS COLLAPSE NAME LIST) and, where necessary, resets TYPBOX. Certain checks then follow, such as, is the identifier preceding a [an array or switch; is the identifier preceding a (a procedure with parameters or a formal procedure zero.

ERRORS

FAIL 48 ; identifier declared locally.

FAIL 17 ; identifier not declared locally.

FAIL 87 ; identifier not switch or label when so required.

FAIL 18 ; Use of undeclared identifier.

FAIL 61 ; (misplaced, or missing procedure name.

FAIL 99 ; Inconsistent use of identifier.

FAIL 38 ; [preceded by other than switch or array.

SECODL

This routine searches the constant area for a given constant. The global variable Q is used to search CODL; where the constant is not found it is entered, thereby updating the pointer CODLP.

TAKE IDENTIFIER

This subroutine is used to find the namelist entry for the current identifier and to compile the correct object program Take Address or Take Value operation.

The subroutine SEARCH fails if the identifier is not found in the Namelist and leaves I pointing to the entry if found. An extra check is made that an identifier occurring in an array declaration is non local. The type of the identifier is then inspected to determine which object program operation should be compiled.

(i) switch e.g. go to S [n]

No operation is required until the switch subscript has been processed.

(ii) label e.g. go to LABEL

GTF or GT is stacked depending on whether the label is a formal parameter or not, together with the label address from the namelist. This will be compiled in UNSTAK at the end of the statement.

(iii) array e.g. a + arrayname [b]

Take Address or Take Formal Address is compiled with the address of the array map.

(iv) type procedure

e.g. FUN :=

A:= a + FUN

In the former case, this is an assignment to a function designator shown by P = 0. If FD = 3 this assignment is outside the procedure body; otherwise FD is set to 1 to show the assignment has been made. IFUN or RFUN is compiled depending on whether this is an integer procedure or real procedure with no parameters.

In the latter case, this is a function call shown by P = 1. An operation to reserve space for the function designator is compiled, followed by CFF or CF, depending on whether this procedure is a formal parameter or not, together with the object program address of the procedure block.

(v) scalar e.g. REAL := int + REAL

The relevant Take Address or Take Value operation is compiled depending on whether P = 0 or 1 and a further distinction is made when the variable is a formal parameter, which may be called by name or value (see TABLE 1).

TABLE 1

P	f[I]	v[I]	type	Object Program Operation	
0	0	-	Integer	Take Integer Address	TIA
0	0	-	Real	Take Real Address	TRA
1	0	-	Integer	Take Integer Value	TIR
1	0	-	Real	Take Real Value	TRR
0	1	0	Integer	Take Formal Address	TFA *
0	1	0	Real		
0	1	1	Integer	Take Stack Integer Address	IFUN
0	1	1	Real	Take Stack Real Address	RFUN
1	1	0	Integer	Take Formal Value Indirect (Take Result Call by Name)	TRCN
1	1	0	Real		
1	1	1	Integer	Take Formal Value	TFV *
1	1	1	Real		

ERRORS

FAIL 41 ; indentifier in bound pair is local

FAIL 22 ; incorrect use of label, go to obscured or missing.

FAIL 111 ; type procedure zero declared with parameters.

FAIL 46 ; assignment to function designator is outside procedure body, or assignment to formal procedure

FAIL 31 ; assignment to switch

FAIL 25 ; non type procedure as function designator

* TFA and TFV are the same interpreter primitive.

TAKE

This subroutine is used to process an identifier or constant that is used in a statement or expression. However, if the current ALGOL section does not contain an identifier or constant, the last delimiter is checked, and if it is] UNSTAK is called to generate INDA or INDR as required.

If the current ALGOL section contains an identifier, it is processed by the subroutine TAKE IDENTIFIER (which uses the P which is a parameter to TAKE).

It should be noted that P = 1 when on the right hand side of an assignment statement, otherwise it is 0,

Where the current ALGOL section contains a constant, P is checked to remove errors such as "2:=", and after a check that a Boolean constant is not used in an arithmetic expression CODL is searched, having the pointer Q set. CONS is set by BCR; to zero if real, to one if integer, and to two if Boolean. The test on CONS determines what instruction to generate before exit.

ERRORS

FAIL 31; constant before := or [

FAIL 45; Boolean constant cannot be used here.

FAIL 30; adjacent delimiters inadmissible.

TYPCHK

This subroutine, called during a for clause, checks the type of the controlled variable (held in LOKTYP) against the type of the current identifier (held in TYPBOX). It generates a conversion if necessary, and resets TYPBOX to the type of the controlled variable.

ACTOP

This subroutine is used in conjunction with PRAMCH to check the legality of an actual parameter and to compile the appropriate object program instruction.

A count of actual parameters in the current procedure call is kept in PRMCOU which is checked, in the routine ')' against the number of parameters in the namelist entry for the procedure name.

If E is set to expression level, this actual parameter could be an expression or a procedure call.

Example

(i) PROCALL (a + Q (s) + b, ...

(ii) PROCALL (a, Q (s), ...

The difference between these cases is shown by the top of the stack; in the latter case the top of the stack is '(' since each actual parameter unstacks back to the '(' of its procedure call; in the case of an expression, the top of the stack will be an arithmetic, relational or logical operator. If this actual parameter is an expression, TAKE and UNSTAK are used to complete the processing of the preceding expression, E is reset to 1, and PRAMCH is called to check that the corresponding formal parameter is a scalar called by value, and to compile a type conversion if necessary.

If this actual parameter is a procedure call, PRAMCH is used to check that the corresponding formal parameter is a scalar called by value.

If E is set to statement level, the last delimiter (LASTDL) is examined to determine between the three possible types for this actual parameter.

Example

(i) PROCALL (a, B[1], ...LASTDEL =]

(ii) PROCALL (a,⟨string⟩, ...LASTDEL = ⟩

(iii) PROCALL (a,b,c .....LASTDEL = ,
or (

In the first case, the top of the stack is used to determine whether this is an array or switch subscript. If the former, PRAMCH is used to check that the formal parameter is a scalar called by value, and to compile a type conversion if necessary. If the latter, the top of the stack is GTS or GTFS and PRAMCH is used to check that the formal parameter is a label called by value.

In the second case, PRAMCH is used to check that the corresponding formal parameter is a string called by name.

ACTOP (Contd)

In the third case, M is tested to determine whether the actual parameter is an identifier or a constant. If M = 1, this is an identifier and the Namelist is searched to find the declared entry. PRAMCH then checks the type of the actual parameter with that of the formal parameter and compiles the relevant object program operations.

If M = 2, this actual parameter is a constant and SECODL is used to find or enter the constant in the constant list, and PRAMCH is used to compile the correct "Take constant" operation.

Calls of PRAMCH:

0	Identifier
1	Constant
2	Expression
3	String
4	Switch call
5	Array subscript or procedure call.

ERRORS

FAIL 6 ; More than 14 parameters

FAIL 5 ; Illegal actual parameter

FAIL 49 ; Blank parameter

ARRAY BD

This subroutine, called by the delimiters , and : , checks the validity of their use. TAKE and UNSTAK are called to complete the processing of the preceding subscript bound expression, at which time the top of the stack should be the opening square bracket introducing the subscripts.

If this is in an array declaration, XX is used to check that (e.g.) :

real array X [a:b: .....

is caught as illegal, and the count of subscripts is incremented in DIM and restacked.

When this is during the use of the array, any necessary conversion is generated before restacking the incremented count of subscripts.

ERRORS

FAIL 66; misused , or : in an expression.

FAIL 50; colon in subscript expression.

FAIL 103; commas or colons wrong in array bounds.

DEC

This subroutine is used by any delimiter that starts a declaration (e.g. integer); it checks the validity of its use and sets up a block if this is the first declaration after a begin.

If begin ALL is on the top of the stack, no action need be taken.

If begin is top, a stopper entry is made in the Name List and the block number updated (begin TR would have already done this). The begin entry is unstacked, and P tested to see if this routine has been called from array. If so, a block entry is compiled into the object program. DECSTA is set to 8 0 before exit.

ERRORS

FAIL 36 ; declaration starts incorrectly

FAIL 54 ; declaration follows statement.

FAIL 63 ; misplaced declarator.

DECL

This subroutine is used to enter all declarative information into the Name List.

After checking that the identifier has not already been declared during this block, name and type are entered. The parameter is then examined for the type, as follows:

0	Scalar.	Space is reserved in the notional data area.
1	Switch.	Space is reserved in CODL (Constant Object Data Load).
2	Procedure.	The current program address is inserted.
3	Array.	ARRCOU is incremented.
4	Label.	Space is reserved in CODL, having inserted the current block number.

ERROR

FAIL 27; declaration without identifier.

ENDSTA

This subroutine is used to complete the processing of a statement. If an input/output statement is terminating the marker is cleared and E set to 1. DECSTA is set to statement level and EXP is used to change the state variable E to expression level if necessary. If E is zero, TAKE is called to generate the correct instruction, and ARITH,E, and EXPTYP are set up to deal with the next statement.

Otherwise, if the current section contains an identifier (which must be a procedure identifier), the Name List is searched to discover whether this identifier is a formal parameter before generating the relevant instruction. EXPTYP is cleared before exit.

ERRORS

FAIL 32 ; constant or identifier other than procedure zero used as statement.

FAIL 20 ; Array element or switch used as statement.

FORCOM

This subroutine is called at the end of a for list element to compile the correct object program operation. TAKE and UNSTAK are used to complete the processing of the preceding expression and the top of the stack (TS) is then used to determine between the various kinds of list element.

EXAMPLE (i) for C V: = a, ... simple

(ii) for C V: = 1 step 1 until n, ... until

(iii) for C V: = a while a <10, ... while

N.B. if FORCOM is called by ',' this could be the delimiter between dimensions in an array subscript used in a list element.

(iv) for C V: = A [1,3] ..... [

In cases (i), (ii) and (iii) the type of the controlled variable is restored from the stack into LOKTYP, and TYPCHK is used to compile a type conversion if necessary. The relevant object program operation is then compiled and a check is made that the top of the stack is for, and an exit is made back to ',' or do.

In case (iv), the number of dimensions is updated and stacked with '[' and a return made to the CENTRAL LOOP, to read the next ALGOL section.

ERRORS

FAIL 96; incorrectly constructed for clause.

FAIL 43; missing] on array element preceding do.

COLLAPSE FORMAL PARAMETERS (FCLAPS)

This subroutine is called from the delimiter ; , such as in the following example:

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

At this stage the name list will look like this

```
block stopper
    P  <----        value of NLP (which was
                       stacked with proc begin)
    a
    b
    c
```

The block stopper is erased, and for type procedures a check made that an assignment has been made to the procedure identifier during the procedure body.

The V bit in the procedure name list entry is cleared, as we are now no longer in the procedure body and recursion cannot occur. WM is set and a warning message is given for every parameter not used, as these parameters are inspected and condensed to one-word entries containing type and whether by value or not.

ERRORS

FAIL 16 ; No assignment to type procedure identifier.

COLLAPSE NAME LIST (NCLAPS)

This subroutine, called from end, resets the Name List Pointer to the position it held before the current block began, and reinstates the previous block number. If any identifier is declared and not used, a warning message is produced, except in the cases of switch names which are ignored, and label names where the warning message is upgraded to a fail. The variable CNL is set on entry so that if more than one label is unallocated each will give a failure message.

ERROR

FAIL 79 ; unallocated label.

real, integer, Boolean

W is set up with the corresponding type, and a check is made that DECTYP is zero. If not, this would mean that we were already in a declaration (as DECTYP is set to zero on ;) such as

begin real integer a.....

DECTYP is then set up with the type from W, and DEC is called with a parameter of 1. This in effect looks back over its shoulder to see whether this is the first declaration in a block (if it is DEC will have to update Current Block Number, DECSTA etc).

ERRORS

FAIL 47; illegal declaration

FAIL 76; misplaced delimiter

array

The subroutine DEC is used as described above, and a check is made on DECTYP. The failure path is exactly like the one above, where we are already in a declaration other than real, integer or Boolean (e.g. "begin string array....").

If DECTYP is set to real, integer, or Boolean, it is further limited to array; if DECTYP is 0, ALGOL specifies that a non-type array is treated as a real array, and DECTYP is thus set. MAMPS (Make Array Maps) is then stacked.

(i) so that inspection of the top of the stack can show us that we are in an array declaration(this is particularly important when the array bound variables are themselves nasty things like procedure calls), and

(ii) so that the relevant object program can be generated at the end of the array bound list (see] flowchart)

ARRCOU becomes zero ready to count array names (see , (comma) which calls DECL (3))

e.g.: begin real array A,B,C,D, [.....

ERRORS

FAIL 72 ; illegal declaration

begin

This routine checks that begin has been used to start a block or statement and sets DECSTA to zero. Note that no block entry is compiled or set up in the Namelist since it is not yet known if this begin starts a compound statement or a block. This work is done in DEC by the first declaration following a begin.

ERRORS

FAIL 60; illegal use of begin i.e. in expression or following :=

do

The subroutine FORCOM completes the object program instructions for the preceding for list element. The object program 'For Statement End' is then compiled and an instruction to update the address following 'For' in the object program to point to the controlled loop. A stack entry is made of for begin to show that this is a for statement, with the second address following 'For' which will be updated to point to the statement following the for statement. The block name is also updated to its highest value.

The 'for clause' is defined in ALGOL as being followed by a Statement (a construct wide enough to include a Block, or another for statement). E is therefore set to 1 (to show that a statement is expected); ARITH and F (the for clause marker) are cleared - the latter was only of use during the processing of the for clause.

else

The object program for the preceding statement or expression is completed by TAKE and UNSTAK. If the top of the stack is 'then S' the subroutine ENDSTA is used to compile the procedure zero call.

e.g. if a < b then procedure else go to label;

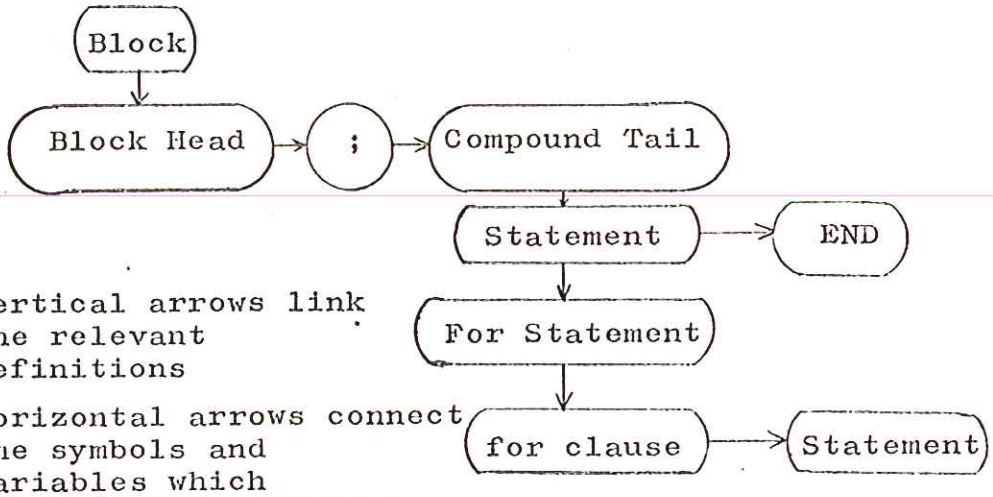
The top of the stack is then tested for 'then S' or 'then E' to differentiate between a conditional statement and a conditional expression. In each case a UJ operation is compiled which will be updated at the end of the statement, and the IFJ operation around the then part is updated.

Finally 'else E' or 'else S' is stacked, (in the case of an expression TYPBOX is stacked to show the type of the then expression), as is the address of the incomplete UJ operation around the else part.

In the case of conditional statements E, ARITH and EXPTYP are reset for the following statement.

end

DECTYP is checked to make sure that this delimiter does not complete a declaration (e.g. begin real a end), and the subroutines ENDSTA and UNSTAK are used to complete the processing of the preceding statement. The top of the stack is then inspected; if end shows that we are ending a for statement (TS = for begin) we must cycle round (having compiled something into the object program using FSEND) and inspect it again, because ALGOL's definition of a for statement is recursive (see diagram) and therefore one end may terminate lots of for statements.



Note:

Vertical arrows link the relevant definitions

Horizontal arrows connect the symbols and variables which together form a definition.

Having unstacked any lurking for begins, the top of the stack should now be a form of begin. If it is not it is a failure. e.g. for end.

Where the top of the stack is begin, this is the end of a compound statement. begin TR signifies that this is the end of a block containing no array (i.e. no block entry compiled), and begin ALL that this is the end of a block containing an array (and therefore there is an unconditional jump to be updated). For the relevant entries that are unstacked here, see DEC.

In the case of a block, the name list is collapsed back to the first entry for this block, as all the variables declared local to this block are now no longer valid.

A test is then made for the end of the program (is stack empty) and if so ENDPRO is accessed. Otherwise BCR is called to extract any comment following end

e.g. end of this routine;

A warning message is emitted if there is a delimiter in this section, which will catch

```
end
x := 1;
```

Final exit is to OUT 2, as BCR will already have recognised

one of the delimiters which terminate an end comment, namely end, else or ;.

ERRORS

FAIL 40 ; Top of stack not a begin

for

After checking that for is used at the start of a statement (using M, F and EXP) DECSTA is set to /o o since this is a statement. A stack entry is made of for together with its address in the object program. This address will be used by do to update the pointer to controlled statement. The current block name is stacked before being updated to the next highest block name for the for block.

The object program compiled contains two markers which will be updated by the addresses to the controlled statement and the statement following this for statement.

The subroutine BCR is used to read the next ALGOL section which should be 'controlled variable := '

e.g. for cv := 1 step 1 until n do _____ ;

TAKE compiles the correct object instruction for the controlled variable and a stack entry is made for the first list element with the type, held in TYPBOX, of the controlled variable. If the controlled variable is type real, the list elements must all be real and similarly when the controlled variable is type integer. The stack entry 'simple' is used to distinguish a simple list element from a 'step' or 'while' element.

Finally, ARITH and E are set for the following list element expression and F is set to 1 to show that this is a for clause.

ERRORS

FAIL 21 ; controlled variable is not a simple variable.

FAIL 44 ; for doesn't start a statement

go to

After checking that the delimiter is used to start a statement, DECSTA is set to /o o to show this is a statement. The delimiter is stacked as GT since it is not yet known if this is a go to label or go to switch statement.

e.g. go to labelname;

go to s [a+b] ;

E and EXPTYP are set up for the following label or switch expression.

ERRORS

FAIL 42 ; go to follows an identifier or constant.

The initial test removes as a failure such phrases as:

```
b := (a+b)  if ...
b := A[b]  if ...
if a > then if ...
b := a+b if ...
```

EXP is then called to set E to zero if the preceding delimiter was := as in

```
a := if ...
```

In the case of E being equal to 1, DECTYP is tested. It is normally zero (i.e. we are not in a declaration) and DECSTA is set to /o o. It may however be set to array, as in:

```
real array A [1 : PROC (if ...
```

In this case of course DECSTA must not be changed. A test is then made to see if this is an actual parameter, and if it is E is set to zero.

If E was not equal to 1, a further test is made to remove constructs such as

```
y + if
```

The stack entry saves the state variables ARITH, E and EXPTYP and stacks if to be checked by the following then. ARITH and E are then set up for the Boolean expression in the if clause.

ERRORS

FAIL 67;	if misused
FAIL 100;	if must not be used after log., arith. or rel. operator.
FAIL 68;	if used in declaration other than array declaration.

Procedure

The subroutine DEC is used to set up a block if this is the first declaration in the block, a stopper is put in the name list, the marker PH is set to show that a procedure heading is being processed, and a call is made to BCR to read the next ALGOL section i.e. 'procedure name' (. A check is made that the preceding delimiter was real integer, Boolean or non-type. Unless the procedure is own code an unconditional jump is compiled to jump round the procedure body and a stack entry is made of proc. begin together with various state variables and the address (PP) of the incomplete UJ operation to be updated at the end of the procedure body.

DECSTA is set to statement level for the procedure statement following the specification part, the number of parameters (PRMCOU) is set to zero and the current block name (BN) is updated for the procedure block.

If the procedure has no parameters (delimiter is;) DECTYP is set to procedure zero and the procedure name is declared in the namelist using the subroutine DECL. The procedure entry operation is compiled, DECTYP cleared and a call is then made to BCR to check whether the procedure body is ALGOL or own code.

If the procedure has parameters, DECTYP is set and the procedure name is declared in the namelist. PRCENT is used to hold the namelist address of the procedure name, and PROC is set to 1. The loop that follows reads a formal parameter, checks that it has not already been declared, enters it in the namelist and updates the count of parameters. When ')' has been read, the procedure entry operation can be compiled and the namelist entry completed with the number of formal parameters; PROC and DECTYP are cleared.

The next call to BCR should fetch ; which terminates the formal parameter part.

Example:

```
                                  f.p. part     value part
integer procedure  F (a, b, c) ;  value a,b;

                                  specification part
                                  real a,c;     array b;
```

Another call is made to BCR to determine whether a value part or the specification part follows. Each identifier in the value part is checked for appearance in the formal parameter list. v:=1 in the namelist entry. When the terminating ';' is read the next ALGOL section is read (specifier,) and the specification part is processed.

This consists of specifiers followed by identifier list(s) e.g. real a,c; There is an inner loop to read each identifier in the list following the specifier, check that it is a formal parameter and complete the namelist entry. If the formal parameter is a switch the number of dimensions is set to 1 but if it is an array or procedure

the number of dimensions or parameters is not yet known and is set to + 15, This number will be updated at the first occurence of this identifier with its parameters, and subsequent occurences must agree.

If the delimiter is ';' this could be the end of the specification part or the end of this specifier list. In the former case, a check is now made that each formal parameter has been specified and if called by value that the type is not switch, string or procedure. The program then compiles the check words for the run time parameter checking.

Finally, a check is made to determine whether the program body is ALGOL or own code. If the latter DECSTA is reset to declaration level, the formal parameters are collapsed and the next delimiter is read to discard the ; 903 ALGOL requires the parameters of formal procedures to be called by name, this is checked in PRAMCH however, when a parameter is found to be of type procedure.

ERRORS

FAIL 101 ; procedure name not followed by ; or (

FAIL 102 ; formal parameter part not followed by ;

FAIL 90 ; wrong delimiter in value or specifier part

FAIL 109 ; constant not allowed in procedure heading

FAIL 65 ; illegal specifier

FAIL 17 ; identifier in specification part is not a formal parameter, or formal parameter not fully specified

FAIL 94 ; string, switch or procedure called by value

FAIL 6 ; more than 14 parameters

FAIL 86 ; procedure inside another declaration

FAIL 85 ; Name list overflows

FAIL 92 ; Identifier not specified

FAIL 88 ; formal parameter not followed by) or ,

step, until, while

The first test is to ensure that these delimiters are only used in a for clause. The arithmetic expression preceding the current delimiter is then completed by the subroutines TAKE and UNSTAK, when the top of the stack should then be simple. The variable G is set up with the quantity stacked with simple, and LOKTYP is also set up. TYPCHK is called to generate any conversion necessary, e.g.:-

```
real A, B ; integer c;
for  A := B + c step
```

and TYPBOX is then set to LOKTYP. The current delimiter is then examined.

In the case of the delimiter step, simple is restacked with a marker to indicate that the delimiter until is required, and an instruction compiled (an example of a compiled for statement is given below). In the case of the delimiter while ARITH is set up for the algebraic expression following.

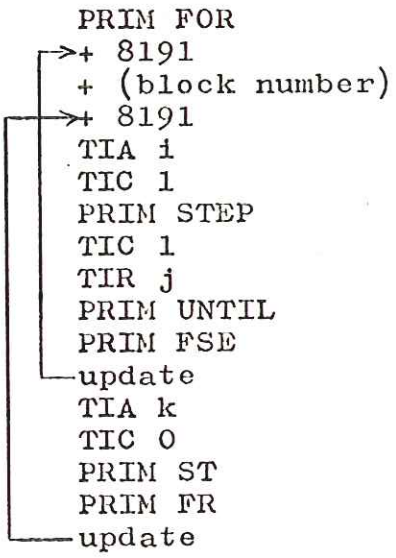
ERRORS

```
FAIL 78 ; corresponding for missing
FAIL 21 ; ":=" omitted from for clause
FAIL 80 ; step, until or while misused in
          for list element.
```

Example

```
begin integer i,j,k ;
    for i := 1 step 1 until j do k := 0
end;
```

generates:



A check is made that we are not already in a declaration and then DECTYP is set to switch. The subroutine DEC is used to set up a block if this is the first declaration in the block. The next ALGOL section is fetched, which should be 'switchname :=', and LABCOU is cleared. DECL is used to declare the switchname in the namelist and make an entry in the label data area (CODL).

The loop is used to process each label in the switch list. (IFIP ALGOL allows only labels in the switch list).

e.g. switch S := TOM, DICK, HARRY;

Each label is declared in the namelist together with the address of the label in CODL and BN is entered in label entry in CODL. At the end of the switch list when ';' is read, the number of labels is entered in CODL and DECTYP is cleared.

e.g. layout of CODL for the above switch declaration.

CODL			Namelist	
n	+ 3	... no. of labels	s	n
n + 1	+ o			
n + 2	BN	... for TOM	TOM	n + 1
n + 3	+ o			
n + 4	BN	... for DICK	DICK	n + 3
n + 5	+ o			
n + 6	BN	... for HARRY	HARRY	n + 5

When the left-hand label is declared, its object program address is placed in the LODL label entry, overwriting the + o.

ERRORS

FAIL 26; no := following switch identifier in switch declaration, or switch misplaced.

FAIL 4; wrong identifier in switch list.

then

This routine first tests that we are at expression level (if not, it fails) and then completes the processing of the algebraic expression following *if* by calling TAKE and UNSTAK. *If* is then unstacked and ARITH, E and EXPTYP are restored. The state variable E is then used to decide whether *then* should be stacked as "*then* S" or "*then* E". The pord pointer (PP) is also stacked, ready to update the IFJ when the delimiter *else* is met.

ERRORS

FAIL 97; *then* in statement

FAIL 69; corresponding if has been omitted or conditional expression without an *else*.

:=

There are three possible uses of the delimiter := as follows:

(i) in a switch declaration, e.g.
switch S := .....

(ii) in a for clause, e.g.
for v := 1 step .....

(iii) in an assignment statement.

The first two are dealt with under switch and for respectively, and this routine merely deals with assignment statements. After checking that we are not assigning to a constant or an expression and that we are not in a declaration, for clause or procedure call, DECSTA and SV set to /o o and TAKE deals with the variable (simple or subscripted) which precedes the delimiter.

As we require all left hand elements in an assignment to be the same type the top of the stack is then inspected to see if it is a :=. If not LHTYPE is set to TYPBOX and the delimiter is stacked, otherwise LHTYPE and TYPBOX are tested for equality and the multiple store function (STA) is stacked.

ERRORS

FAIL 28 ; := preceded by constant or used inside an expression.

FAIL 52 ; := must not appear in actual parameter list, or in a type or array declaration.

FAIL 7 ; := appears in a for statement and other than in assignment to controlled variable.

FAIL 112; Different types on Left hand side of an assignment.

;

Most of the processing for a ; in a declaration is dealt with in that declaration

e.g. switch or scalar do their own, and array declaration is done in RSBRAK

STATRM is then called to complete the processing of the read or print statement. ENDSTA is used to complete the processing of the constituents of the statement otherwise, and the top of the stack is then tested.

If this is for begin, the one or more for statements are completed. If it is proc begin, this is unstacked, the variables stacked with it are restored, and COLLAPSE FORMAL PARAMETERS is called to condense the formal parameter entries in the Name List to two parts — type, and whether the parameter is called by name or value.

If the top of the stack is any other form of begin, the routine is finished.

ERRORS

FAIL 53	;	TS not a form of begin.
FAIL 93	;	Declaration ends incorrectly (STATRM)
FAIL 20	;	Array or Switch element as statement (ENDSTA)
FAIL 32	;	Constant or other than a non-type procedure zero as a statement (ENDSTA)

Arithmetic Operators (+ - x / div)

As the treatment of these operators differs only slightly their stacking priority is recorded and then they share the routine. Their validity is checked using EXP which sets E to expression level. If the operator is not preceded by an identifier or constant and is not a closing bracket it is stacked with its type and associated priority, except for special cases unary plus which is ignored, and unary minus which is noted for special action.

Otherwise the identifier or constant is dealt with and TAKE and UNSTAK are called, or in the case of the closing bracket just UNSTAK.

ERRORS

FAIL 30 ; Adjacent arithmetic operators.

FAIL 57 ; Adjacent operators inadmissable.

Relational Operators (<>≤≥≠≡)

After checking the validity of the use of the operator, TAKE and UNSTAK are used to process the preceding identifier or constant, and to unstack any operators with priorities greater than or equal to the priority of the current delimiter.

The current operator is then stacked with its stack priority and the type of the preceding variable or expression. E is set to expression level.

ERRORS

FAIL 58 ; illegal use of relational operator.

Logical Operators ($\supset \vee \wedge \neg$)

The stack priority of the present operator is stored and a check is made on the use of this delimiter. After setting E to expression level, further checks are made if the current operator is $\neg$ Except in that case, TAKE and UNSTAK are used to process the preceding identifier and to unstack any operators with priorities greater than or equal to the priority of the current operator. Finally the delimiter is stacked with its priority.

ERRORS

FAIL 58 ; logical operator misplaced

FAIL 59 ; illegal use of $\neg$

[

This routine deals with the use of this delimiter for a subscripted variable, a switch designator or as the start of a bound pair list in an array declaration.

Examples: (i) ..... + TABLE [2,6]
(ii) go to SWITCHLIST [4]
(iii) array TABLE [1:p, -3: n]

If DECTYP is clear, this can be a subscripted variable or a switch designator, and DECSTA is set to statement level. TAKE is used to process the switch or array identifier and leaves I pointing to the namelist entry. If the preceding identifier is type array, a stack entry is made of [together with the current values of the state variables. E, ARITH and EXPTYP are then set for the following arithmetic expression.

If the preceding identifier is type switch, a test is made that the preceding delimiter is go to unless this is an actual parameter.

Example: go to S [4] or FUNCTION (S[4],p)

GTFS or GTS is stacked depending on whether this is a formal parameter reference or not, together with address in the namelist entry. The current delimiter is then stacked.

If DECTYP is set, this delimiter is being used to start a bound pair list or to start a subscript expression inside a bound pair list.

Example: array TABLE [1 : LIST [3], 2 : n]

The former case is shown by the fact that the top of the stack is MAMPS (array map) and DECL is used to declare the array name in the namelist. '[' is stacked with a dimension count of 1, a marker of 0 to signify lower bounds and I to give the namelist address of the arrayname.

In the latter case, E is set to expression level and TAKE is used to process the preceding array name, before [is stacked.

ERRORS

FAIL 33 ; Opening square bracket follows closing bracket.

FAIL 73 ; Opening square bracket not preceded by identifier.

FAIL 24 ; Switch designator not a parameter or preceded by go to

The preceding expression is completed using TAKE and UNSTAK. The top of the stack then indicates whether this is an array or switch subscript expression, or an array bound pair list.

In the former case the top of the stack is [and the TYPBOX is tested to determine whether the subscript expression is type integer or real. If real, a conversion to integer must be compiled. The values of the state variables as at [are then restored from the stack. The number of dimensions counted on the stack with [is checked with the array declaration in the namelist. If this is -1 in the namelist (formal parameter specifier) the number of dimensions is now entered.

Example: procedure P (b) ; array b ;
A:= b [1,3,n] ;

When the array b is used, it is found that the number of dimensions is 3 and can therefore be placed in the namelist entry for b.

If this is a switch subscript, there must be only one dimension. Finally, if this is an array subscript, INDR is compiled in the case of an expression and IND is stacked in the case of a statement since it is not yet known if INDA or INDR is required.

Example: A [2] : = B [1, n] : = ...

In the latter case, when this is an array declaration, the number of dimensions is placed in DIM, a lower/upper bound marker in xx and I is restored to the array namelist entry. A check is made that the bound pair ended with an upper bound (xx = 1), and E and ARITH are set for the following declaration or statement. The top of the stack should then be MAMPS and this is compiled with the number of dimensions and arrays (ARRCOU). The namelist entries for each array are then updated with the number of dimensions and the address of the array map in the object program. Finally, the shared map is compiled.

Example: array A,B,C [p;q, s:t]

ARRCOU will be 3 and number of dimensions will be 2.

ERRORS

FAIL 74 ; unmatched closing square bracket.
FAIL 51 ; wrong number of dimensions in array subscript
FAIL 95 ; more than one dimension in switch subscript
FAIL 75 ; upper bound missing from bound pair
FAIL 23 ; incorrect array declaration.

DECTYP is used to decide whether the current identifier follows a label or a lower bound in an array declaration.

In the latter case, ARRBND processes the current ALGOL section.

In the former case, EXP is used to check the validity of the use of the label, and DECSTA is set to statement level (as one cannot label a declaration without an intervening begin). The Name List is then searched for the label (which must have been declared in a prior switch list declaration), and the label entry in Label Object Data Load (CODL) should have an address part of zero. If not, this label has been met on the left-hand side of a colon twice and this is an error.

The next test is to see whether this label has been declared in the current block. If it hasn't, it is an error except in the case of procedure definitions or for clauses (see example below). In these cases, the existing entry in CODL is cleared and a new entry made.

ERRORS

FAIL 29; : in type or switch declaration.

FAIL 9; label used twice on left-hand side.

FAIL 48; misused identifier.

```
Example:  begin real c;  switch S:= LABEL;
            procedure P (q,r,s); integer q,r,s;
              LABEL:  begin q ; = r + s;
                        go to LABEL

            end;
```

If the first begin sets the block number to (say) 52, LABEL is entered by its declaration in Block 52. But procedure P resets the block number to 53, and when LABEL is met on the left-hand side, the block numbers do not correspond.

This routine deals with various uses of the delimiter , .

i) INOUT (1) is called to deal with a read or print list comma.

ii) If we are in a procedure call (PROC = 1) the subroutine ACTOP is used to process the current ALGOL section.

iii) If the for clause marker is set, the subroutine FORCOM is used to distinguish between the use of comma between for list elements in which case PRIM DO is compiled after a typecheck, or between subscript expressions of a variable used in a for list element. The latter case does not return from FORCOM.

The former case then stacks the simple for element again, sets the state variable ARITH to 1 and exits.

iv) If DECTYP = 0 the current delimiter is being used between subscript expressions, and is processed by the subroutine ARRBND.

v) Finally, the top of the stack is used to differentiate between various uses of a comma in an array declaration.

If the top of the stack is MAMPS, the comma is between array identifiers viz:

real array A,B, ...

and the identifier is entered in the Name List using DECL.

Otherwise the comma is being used between subscript expressions, and ARRBND processes the current ALGOL section.

ERRORS

FAIL 96; incorrect for clause (FORCOM)

FAIL 77; incorrect use of comma, or missing identifier in array declaration.

After the initial checks to catch subscripted constants and (immediately following a closing bracket DECTYP is tested to see whether we are in array declaration or in a statement. The state variable M will tell us whether this delimiter is being used as an expression bracket or in a procedure call.

In the case of an array declaration, a failure is indicated if E is set to statement level, unless PROC shows that an actual parameter is being processed, e.g.

real array A [1: PROC (a, PROC2(....

or real array A [1: PROC (a,(b + c ....

The former example will cause M to be set at 1, and the latter zero, which is the subject of the next test. Where this is a procedure call, the flowchart joins with the path where DECTYP was zero and this was a procedure call (which has already called EXP to change E to expression level if necessary, and set EXPRES in the case of a read or print statement e.g. read reader (1)...)

The case of "go to S (..." is failed by the next test on EXPTYP; and the procedure name is then searched for in the Name List. If we are in an array declaration, we check that the array bounds are not local (illegal ALGOL), before discovering whether this is a type procedure. If so, space is reserved on the run-time stack (by compiling PRIM UP) for the result of the type procedure. Various variables are then stacked with the delimiter, and these variables are then set up to deal with the actual parameters of this procedure call.

In the case of expression brackets used in statements, the delimiter is stacked, and PROC set to zero while processing the constituents of the bracket.

ERRORS

FAIL 61 ; misplaced
FAIL 82 ; (must not appear in a type declaration
FAIL 62 ; function designator as subject of "go to"
FAIL 41 ; array bounds must not be local
FAIL 25 ; non type procedure as function designator in expression

)

The state variable PROC indicates whether this delimiter is an expression bracket or a procedure call bracket.

If it is an expression bracket, we first check that we are in fact within an expression. This kills "a + (b)". The translation of the preceding expression is completed using TAKE and UNSTAK, after which the top of the stack should be the delimiter (. This is unstacked and discarded, having reset PROC to the stacked value.

If it is a procedure call bracket, ACTOP is called to complete the translation of the preceding parameter. PROCPO is saved in I after the top-of-stack test, and the stacked state variables restored. A test is now made to see whether the count of parameters in the call is equal to that in the declaration; if not, it may be that this procedure is itself a formal parameter. If so, the specifier in question has not told us how many parameters are required, and we must fill the count in from this call. However, we must first check that this count has not been filled in by some previous use, if so it must fail as in the last line of the following example.

```
real procedure JIM (a,b); integer a ; real
                                  procedure b;

    begin integer q,r,s;

        a : = b (q,r,s);
        a : = b (q,r);
```

The test on f [I] determines what sort of instruction to compile for this procedure call. Finally, if E is set to statement level a check is made that the next delimiter is an "end-of-statement" delimiter, otherwise in the expression case TYPBOX must be set.

ERRORS

FAIL 81;	misused) other than in expression.
FAIL 82;	unmatched closing round bracket.
FAIL 5;	illegal parameter list.
FAIL 51;	incorrect number of parameters.
FAIL 111;	incorrect number of parameters in use of formal procedure.
FAIL 84;	wrong delimiter after procedure statement.

⟨ (string opening quote)

This routine checks that this string is being used as an actual parameter, or in a print statement, and then compiles an unconditional jump around the string, which will be updated when the last character in the string is read. There is a count of nested string quotes, and the characters are packed for the object program. When the final closing string quote is read, the UJ operation is updated and '⟩' is stored as the last delimiter. If a parameter case the following call to BCR should fetch ' ,' or ')', and ACTOP is then called. If a print statement INOUT is compiled, and the delimeter which may be end or else also decides the following action.

Example PROCALL (a, ⟨string⟩, ⟨otherstring⟩)

After compiling the actual operation for the string, ACTOP is used to check that the corresponding formal parameter is specified as string called by name. E is set to statement level and if the current delimiter.is ')' exit from this routine is to) Entry 2 to complete the processing of the procedure call.

ERRORS

FAIL 5 ; string is not the complete actual parameter

FAIL 105 ; wrong delimiter following string.