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.

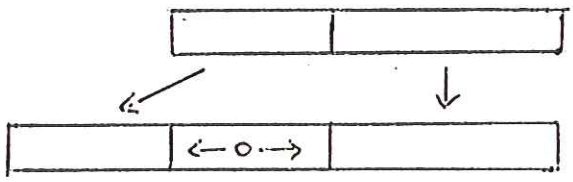
The third word wd 3 contains in its function part a 1 or a 2. If it contains a 1 then the procedure must be on the library tape and the address part of wd 3 must point to the name e.g.

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

1.	INTRODUCTION	3
2.	LIMITATIONS IMPOSED ON IFIP SUBSET ALGOL	4
3.	THE DATA STORE	5
4.	THE CHARACTER SET	10
5.	PROGRAM STRUCTURE	11
6.	TYPE HANDLING	12
7.	NOTATION	20
8.	REFERENCE LISTS	22
9.	NOTES ON TRANSLATOR LISTING	32

Routines

CENTRAL LOOP	33
FAIL	34
TAKCHA	35
IDENTIFIER	36
EVALNA	37
NUMBER	38
BCR	39
UNSTAK	40
EXP	41
PRAMCH	42
SEARCH	47
SECODL	48
TAKE IDENTIFIER	49
TAKE	51
TYPCHK	52
ACTOP	53
ARRAY BD	55
DEC	56
DECL	57
ENDSTA	58
FORCOM	59
COLLAPSE FORMAL PARAMETERS (FCLAPS)	60
COLLAPSE NAME LIST (NCLAPS)	61
real, integer, boolean	62
begin	63
do	64
else	65
end	66
for	68
goto	69
if	70
procedure	71
step, until, while	73

Routines continued

switch	74
then	75
:= (BECOMS)	76
; (SEMICO)	77
arithmetic operators	78
Relational operators	79
Logical operators	80
[(LSBRAK)	81
] (RSBRAK)	82
: (COLON)	83
, (COMMA)	84
((LRBRAK)	85
) (RRBRAK)	86
‹ string opening quote	87

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)

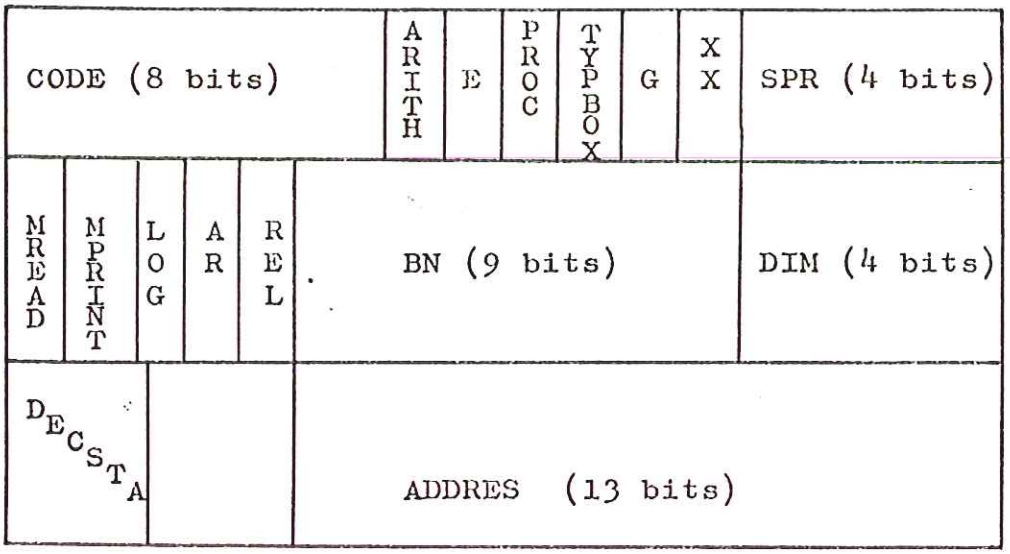
PARAMETER ENTRY

ZERO	V	TYPE (12 bits)	

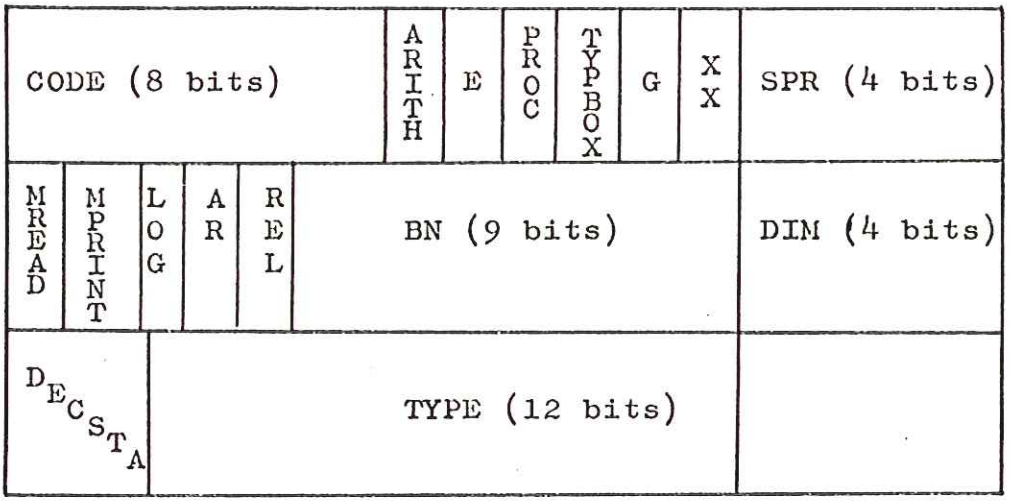
3.3 The Stack

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:-

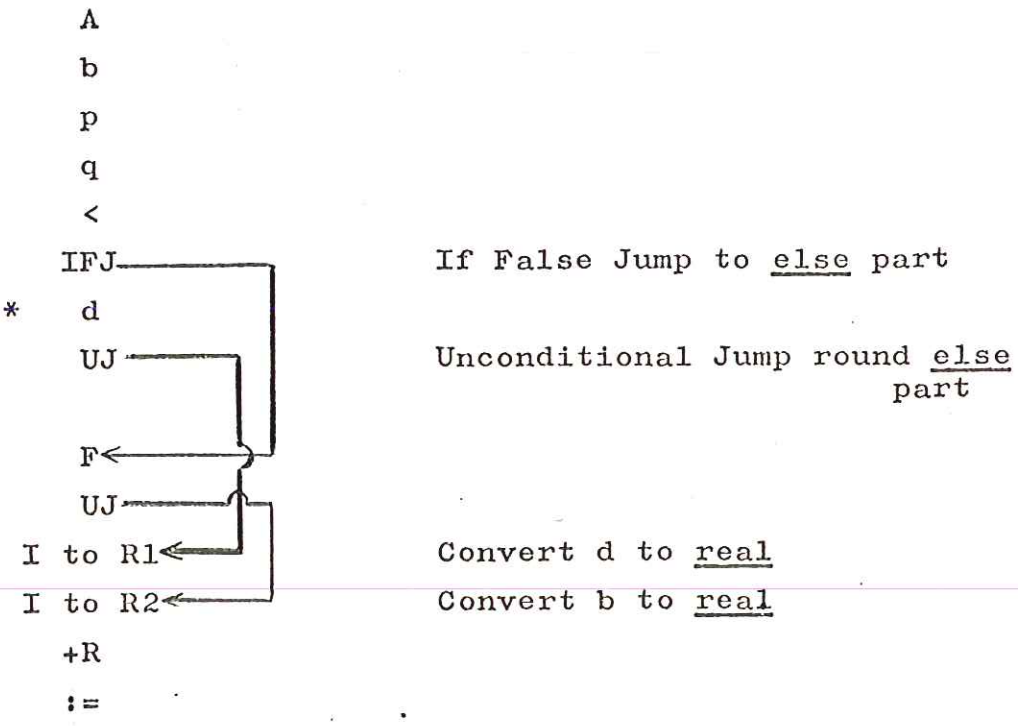
```
a
b
D
e
I to R1        convert e to real
- R
I to R2        convert b to real
+ R
R to I         convert b + (D-e)
:=               to integer
```

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.

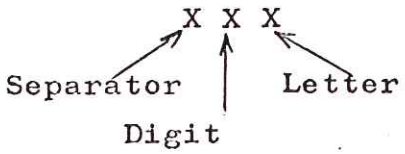
A subscripted item has the subscript value enclosed in square brackets. If an identifier is enclosed in round brackets this may be taken as 'contents of'. e.g.

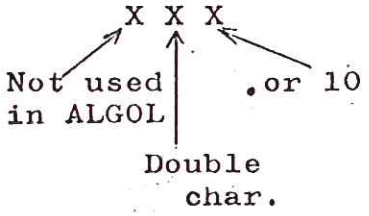
$$(NLP) := \overset{-1}{BN}$$

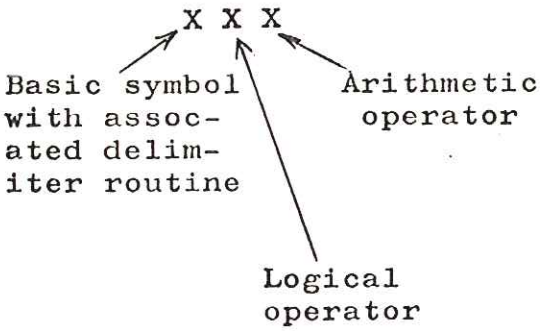
is putting a block stopper in the name list. -1 is put where the name list pointer points and the Block Number goes in the next word, Replacing the Global Block Number; BN := (NLP+1) rather than BN := NLP[1].

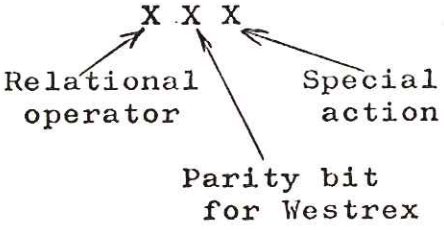
To make stacking easier, the Global variables take the values corresponding to the stacked position. E for example is stacked as the 9th significant bit of word one, and consequently the variable is carried about as 0 or +256 in the coding. The flowchart however, refers to E as 0 or 1. It is thought 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
```