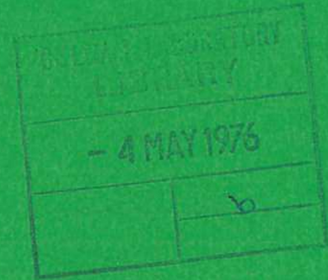
CLM - P 450

This document is intended for publication in a journal, and is made available on the understanding that extracts or references will not be published prior to publication of the original, without the consent of the authors.

CLM - P 450


- 4 MAY 1976

UKAEA RESEARCH GROUP

Preprint

TRANAL
A PROGRAM FOR THE TRANSLATION OF SYMBOLIC ALGOL I INTO SYMBOLIC ALGOL II

K V ROBERTS
L G KUO-PETRAVIC
M PETRAVIC

CULHAM LABORATORY
Abingdon Oxfordshire

1976

The information contained in this document is not to be communicated, either directly or indirectly, to the Press or to any person not authorized to receive it.

Enquiries about copyright and reproduction should be addressed to the Librarian, UKAEA, Culham Laboratory, Abingdon, Oxon. OX14 3DB, England.

CLM-P450

TRANAL
A PROGRAM FOR THE TRANSLATION OF SYMBOLIC ALGOL I INTO SYMBOLIC ALGOL II

by

K V ROBERTS

UKAEA Culham Laboratory, Abingdon, Oxon, UK.
(Euratom/UKAEA Fusion Association)

L G KUO-PETRAVIC

Atlas Computer Laboratory, Chilton, Didcot, Oxon., and St Hilda's College, Oxford, Oxon, UK.

and

M PETRAVIC

Dept of Engineering Science,Oxford, Oxon, UK.

(Submitted for publication in Computer Physics Communications)

December 1975
hp

Title of program: TRANAL

Catalogue number:

Program obtainable from: CPC Program Library, Queen's University of Belfast, Belfast BT7 1NN, N Ireland.

Computer: ICL 4-70; Installation: UKAEA Culham Laboratory

Operating system: ICL Multijob

Programming languages used: ECMA Algol, with ICL System 4 I/O procedures

High-speed store required: 42976 bytes + Stack

No. of bits in a word: 32

Overlay structure: None

No. of magnetic tapes used: None

Other peripherals used: 2 line files for output

No.of cards in combined program and test deck: 700

Card punching code: EBCDIC

Keywords: Symbolic Algol, Translator, Magnetohydrodynamics, Vector notation.

PROGRAM SUMMARY

Nature of problem

Two symbolic styles of Algol 60 programming have previously been developed for the expression of partial differential equations in vector form: SAI [1] which solves the equations directly but executes slowly because of the large number of nested procedure calls involved and SA II [2] which is used to generate an optimized version of the program in a chosen target language. Conversion from SAI to SAII has previously been carried out by hand and the present program TRANAL is intended to perform this process automatically. It is written in Algol 60 and illustrates the usefulness of Algol for the manipulation of physics equations. The main SA I & II program packages will be published in due course.

Method of solution

Each SA I equation is read character by character and converted by recursive syntax analysis into a tree structure of the type illustrated in Fig.3. It is then output in SA II form by an appropriate set of rules.

Restrictions on the complexity of the problem

The number and complexity of the equations can be increased by changing the symbolic parameters in #1.2 of the program. TRANAL is written in ECMA Algol 60 and conversion to most computer systems should only entail changing a few system-dependent I/O procedures in # 6. The operators currently allowed are the basic operators +, -, *, / together with arbitrary unary and binary function operators such as CURL(A) and CROSS(A,B). Limited error diagnostics are provided, and the working of the program can be examined on a second output channel by including the symbol '$' in the input deck.

Typical running time

With diagnostics switched off, the Test Run which converts the SA I equations of Fig.1 to the SA II equations of Fig.2 requires 3½ secs on the Culham ICL 4-70.

References

[1] K V Roberts and J P Boris. Journ.Comp.Phys. 8, 83 (1971).

[2] M Petravic, G Kuo-Petravic and K V Roberts. Journ.Comp.Phys. 10, 503 (1972).

1. INTRODUCTION

In the first paper [1] of this series on Symbolic Algol it was suggested that the many successes of mathematics and theoretical physics can be linked rather closely to the availability of a widely-accepted notation which is elegant, compact and precise. One might expect that comparable successes for computational physics would have to await the development of an equally lucid, equally powerful symbolic method for expressing algorithms and programs directly. Existing high-level languages do not yet provide such a symbolism and a substantial amount of detailed hand coding is still required.

As an illustration of what might be achieved in practice, Roberts and Boris [1] explored a style of programming in Algol 60 which they termed Symbolic Algol I (SAI), and applied it to the solution of sets of non-linear partial differential equations in vector form. An example is the magnetic advection and diffusion equation in the TRINITY magnetohydrodynamics program [1] :

$$\frac{\partial \underline{B}}{\partial t} = \nabla \times (\underline{V} \times \underline{B}) + \eta \nabla^2 \underline{B} \qquad (1)$$

where $\underline{B}$ is the magnetic field vector, $\underline{V}$ the fluid velocity, and η the resistivity. In SAI this can be written as a symbolic difference equation using a notation quite similar to that of ordinary mathematics:

```
NEWB:= B+DT*(CURL(CROSS(V,B)) + ETA*DELSQ(B));                    (2)
```

In both formulae (1) and (2) the notation is powerful enough to omit explicit reference to the dimensionality, coordinate system, vector components and spatial position, and in addition, formula (2) does not need to refer explicitly to the type of difference scheme that is being used. All these aspects can be handled by 'hidden' global variables and lower-level procedures.

Although SAI is convenient for rapid program development using small mesh sizes, this elegance of notation is marred for production runs by the fact that programs in SAI are quite slow to run because of the large number of nested procedure calls that require to be executed at run time. To some extent this deficiency might be alleviated by modifying a standard Algol 60 compiler to speed up the type of procedure call that SAI uses. Altering compilers is rarely practicable nowadays, and therefore we have turned our attention to existing software and shown [2] that, by a simple but somewhat cumbersome transformation of the SAI equations to another style of programming termed Symbolic Algol II (SAII), very efficient code in almost any Assembler or high-level language may be generated. The SAII equivalent of (2) is

```
EQUATE(NEWB,SUM(B,MULT(DT,SUM(CURL(CROSS(V,B)),MULT(ETA,DELSQ(B))))));
                                                                   (3)
```

which is evidently in 1-1 correspondence with (2). The generator program is written in Algol 60 and it incorporates problem-dependent statements such as (3) as an integral component as explained in ref.[2].

Conversion of formulae such as (1) or (2) to the notation (3) can actually be done quite quickly by hand since only a few lines are involved, and in practice this is how the SAII technique has been employed for solving physics problems [3]. Nevertheless the large number of brackets appearing in version (3) appears awkward, and in a fully-developed problem-solving system one would prefer to input the equations in a form such as (2) or perhaps alternatively

DB/DT = CURL(CROSS(V,B) + ETA*DELSQ(B)   (4)

which is equivalent to (1) and might then be automatically converted to (2) or (3) as required.

The purpose of this paper is to describe how a translator program, TRANAL, can be used for this type of conversion. TRANAL is written in Algol 60 and an annotated listing together with a commentary is given in Appendix A.

As a matter of practical convenience we have simplified the input equations to the form shown in Fig.1, i.e. by omitting the colon in (2), (if present it is ignored), and allowing the variable name on the LHS of the equation to be identical (where appropriate) to the corresponding name on the RHS. This differs slightly from legal Algol 60 because in SAI the names of the physical dependent variables on the RHS represent real procedures [1] and therefore cannot also appear on the LHS of an assignment statement. Any other minor variation of notation can readily be incorporated by suitable modifications to the translator program.

2. CHOICE OF LANGUAGE

It is clear that the transformation from (2) to (3) involves writing the operators (here we include the symbol :=) in their prefix notation, i.e.

A+B → SUM(A,B)
A:=B → EQUATE(A,B)   (5)

while at the same time preserving all the operator precedence implied in the original equation. The first version of the translator to be developed, TRANSTAG [4,5] used the STAGE 2 macro-processor [6]. This string processor has been much favoured by some systems analysts, but it can reasonably be argued that since the generator program [2] which uses SAII as input is written in Algol 60, it is desirable to have the SAI translator in Algol 60

too. This would then alleviate the necessity for the potential user to have STAGE 2 available at his installation and to familiarize himself with it before being able to take full advantage of Symbolic Algol programming techniques.

A further reason for choosing Algol 60 rather than STAGE 2 is that the Algol coding is considerably more intelligible, as a comparison between the listing of TRANAL in Appendix A and the listing of TRANSTAG reproduced in ref.[5] should make clear. Although the latter is very thoroughly commented and is supplied with explanatory tables, nevertheless it is quite difficult to understand. The Algol 60 version is also much faster, taking only $3\frac{1}{2}$ seconds on the Culham ICL 4-70 to process the 4 TRINITY equations of Fig.1 compared to 18 minutes on the ICL KDF9 using TRANSTAG, the ratio of the machine speeds being of order 3:1. Therefore we feel that TRANAL is to be preferred and should be published here as an example of a potentially important field of application of the Algol 60 language to computational physics problems.

Both the STAGE 2 program TRANSTAG and the present Algol 60 program TRANAL make good use of recursion for syntax analysis, but while TRANSTAG basically manipulates with strings, the present program mainly employs pointers and tables and is consequently more economical in storage space and time. The availability of recursion gives Algol 60 an important advantage over other higher-level languages such as Fortran for this type of work, but a definitive comparison with other more sophisticated string-processors must be left for the future. In any case it is suggested that Algol 60 is a useful weapon in the computational physicist's armoury.

The program package described in this paper has been developed on the ICL 1906A at the SRC Atlas Computer Laboratory, Chilton and then finalized for publication on the ICL 4-70 at Culham. It is punched in the ECMA hardware representation. Only the control cards and a few short procedures in ≠6 of the program need be changed to make it run on a different type of computer which accepts this representation, while in most cases it should be possible, if necessary, to convert the hardware representation to a different form in a methodical way using a context editor.

3. DOCUMENTATION

Even though the package is written in high-level language the listing of the card deck would still be quite difficult to follow on its own. This is partly due to awkwardness of the ECMA hardware representation, e.g.

```
LEFT'⟨'KT'⟩':= 'IF'MNUS'⟨'KT'⟩' 'EQ' 1
    'THEN' DUMOPD 'ELSE' POPD;                          (6)
```

is harder to read than the equivalent reference representation

```
LEFT[KT]:= if MNUS[KT]=1 then DUMOPD else POPD;        . (7)
```

Nevertheless even the Algol 60 reference representation needs considerable annotation to make its working clear: it is certainly not true that high-level languages are self-documenting for this type of program and this is hardly surprising since ordinary mathematics usually needs to be explained by a substantial proportion of text. Rather than incorporate the annotation as comments and indexes in the actual code we believe that it may be more illuminating to document a program of this kind in two parallel columns as shown in Appendix A. The right-hand column is a typed version of the Algol 60 code in a form which is as close as practicable to the reference representation while the left-hand column contains a 'program commentary'. The two columns are lined up horizontally so that each can be used to help interpret the other. In practice this means leaving considerable gaps in the code since this is more concise than the annotation; however such gaps do not occur to the same extent in the punched-card version.

The program commentary is supplemented by the diagrams and tables which appear in this paper together with a limited number of headings and comments in the code itself which serve as pointers.

4. INPUT AND OUTPUT DESCRIPTION

The data for Test Run 1 uses the 4 equations of TRINITY shown in Fig.1. Using the second equation as an example we see that each equation consists of:

(i) A name on the LHS.

(ii) An '=' sign

(iii) An expression on the RHS consisting of elements termed:

operands	RHO, V, DT, TEM, B, NU, RHO
numbers	2
operators	( ) + - * / ,
unary function operators	GRAD, DIV, DELSQ
binary function operators	TEN

(iv) A terminating semi-colon.

The corresponding SAII output from Test Run 1 is shown in Fig.2 and diagrammatically in tree form in Fig.3. The identifier EQUATE is an Algol 60 procedure and all other leaves and nodes in Fig.3 correspond to typed

procedures (which for consistency with SAI are taken as real) except for the number '2'. These SAII procedures are used to generate the required code as explained in ref.[2]. For this purpose the operators +, -, *, / have to be converted into binary real procedures SUM, DIFF, MULT, QUOT respectively. A minor complication arises because the operator - can also appear in a unary role, either after the '=' sign or after an opening bracket or comma, e.g.

$$X = -A*(-B+C(D,-E)) \quad . \tag{8}$$

The operator + can also appear in this role although normally it would be omitted. To allow for this situation an operand BLANK is inserted as in Figs.2 & 3 and this can then be arranged to modify the code generation in an appropriate way.

Both TRANAL and the SA II generator are intended as research tools and can readily be modified to suit particular requirements. At present only limited error diagnostics are provided, since they would necessarily make the working in Appendix A more difficult to follow and therefore might not be appropriate in this paper. Any input text which is preceded by '&' and followed by '!' is transmitted to the output channel unchanged, these control symbols themselves being deleted. This is convenient when ordinary Algol 60 statements or comments are to be included in the SA II program.

The standard input/output channels are labelled NIN/NOUT respectively. Error diagnostics appear on channel NOUT; if diagnostics concerning the working of the program are required they are switched on by including a $ in the input deck, and appear on a channel NDIAG which is normally different from NOUT. Some of the current restrictions on the input format are:

(a) There should be a variable with a name not exceeding MAXLHS=10 characters on the LHS of the equation. Change the dimension MAXLHS of the array LSTORE for longer names.

(b) Each equation should be terminated by a semi-colon.

(c) Blank characters and lines are ignored.

(d) The total number of equations should not exceed MAXEQU=20. Change this value for more equations.

(e) The input deck should be terminated by a £ sign.

(f) The total number of operands and operators in an equation should not exceed MAXS, currently set to 200. If more are required increase the value of MAXS, and possibly the dimension MAXSTO of the array ASTORE should also be increased.

(g) Only unary and binary function operators are allowed.

5. METHOD OF OPERATION

The operation of TRANAL is divided into three sections:

1. procedure INPUT. Read in strings of the form shown in Fig.1, one equation at a time, and store information about the operands and operators in tables. Each operand or operator on the RHS is represented by the sequence number N of its initial character, blanks being ignored.

2. procedure CREATE. Analyse each input string to produce the equivalent of a tree of the form shown in Fig.3. With each node N are associated two array elements LEFT[N] and RIGHT[N], which indicate the nodes or leaves to which it is connected. A unary function operator has a dummy leaf at the left, indicated by a dashed line in Fig.3. Procedure CREATE is called recursively, each opening bracket causing a new copy to be entered, and a closing bracket causing a corresponding return.

3. procedure OUTPUT. Generate SAII output of the form shown in Fig.2.

These three procedures are called in turn by procedure MAIN.

5.1 Input

The SA I equations are presented as data to the program, and the characters are read in one by one by the statement

I:=NEXTCH . (9)

This reads the next character from input channel NIN and assigns the integer corresponding to the internal code value to the integer I. (This code is EBCDIC on the ICL 4-70). Various special characters are checked, and the name on the LHS is stored character by character in the array LSTORE, processing being terminated when an '=' sign is encountered.

The operators and operands on the RHS are then stored in the array ASTORE, using ';' as a delimiter for separating the names, numbers or symbols. The address MA in ASTORE of the symbol or the first character of a name or number is stored in the array S.

In order to distinguish an operator from an operand, the address of an operator is stored as a negative number - MA where MA is the actual address. There are two additional arrays FUNC and MNUS which are of the same dimensions as S and carry subsidiary information about operators which have no operands on the left :

Table 1

Input analysis for the velocity equation.

N	S[N]	ASTORE	FUNC[N]	MNUS[N]	LEFT[N]	RIGHT[N]
1	- 2	(				
2	3	RHO				
3	- 7	*			2	4
4	8	V				
5	- 10	+			3	7
6	11	DT				
7	- 14	*			6	44
8	- 16	(				
9	- 18	-		1	D	10
10	- 19	GRAD	1		D	15
11	- 24	(				
12	25	RHO				
13	- 29	*			12	14
14	30	TEM				
15	- 34	+			13	22
16	- 35	DOT	1		18	20
17	- 39	(				
18	40	B				
19	- 42	,			(18)	(20)
20	43	B				
21	- 45	)				
22	- 47	/			16	23
23	48	2				
24	- 50	)				
25	- 52	-			9	26
26	- 53	DIV	1		D	36
27	- 57	(				
28	58	RHO				
29	- 62	*			28	30
30	- 63	TEN	1		32	34
31	- 67	(				
32	68	V				
33	- 70	,			(32)	(34)
34	71	V				
35	- 73	)				
36	- 75	-			29	37
37	- 76	TEN	1		39	41
38	- 80	(				
39	81	B				
40	- 83	,			(39)	(41)
41	84	B				
42	- 86	)				
43	- 88	)				
44	- 90	+			25	46
45	91	NU				
46	- 94	*			45	47
47	- 95	DELSQ	1		D	50
48	-101	(				
49	102	RHO				
50	-106	*			49	51
51	107	V				
52	-109	)				
53	-111	)				
54	-113	)				
55	-115	/			5	56
56	116	NEWRHO				

D denotes a dummy operand with N-value NMAX+1. The operator COMA has the same LEFT and RIGHT entries (shown bracketed in the table) as those of the corresponding function operator, but does not itself appear in the tree structure of Fig.3.

(a) FUNC(N) = 1 if S(N)=-MA, and MA is the address of a function operator such as CURL or CROSS.

(b) MNUS(N) = 1 if S(N)=-MA, and MA is the address of a + or -, which must then either be the first character on the RHS or be preceded by an open bracket or a comma.

At the end of this section the entries for the 4 arrays ASTORE, S, FUNC and MNUS will have been completed. An example for the velocity equation is shown in Table 1. Numerical operands are dealt with by procedure GWORD during the output stage.

5.2 Syntax analysis

After the RHS of a complete equation has been read in, syntax analysis begins by a single call to the recursive procedure CREATE. The operators and operands are represented by the sequence number N which is used to manipulate the operator and operand stacks PSTACK and DSTACK respectively. An operand goes on to DSTACK without comparison, while an operator only goes on to PSTACK after comparison with the priority of the existing operator ITOP at the top of the stack. The operator priorities are given in Table 2.

Table 2

Operator Priorities

Operator	Priority
,	1
+	2
-	2
*	3
/	3
functions	10

If the priority of ITOP is greater than equal to that of the incoming operator, then RIGHT[ITOP] and LEFT[ITOP] in Table 1 are filled with the two uppermost entries on the operand stack, and ITOP itself is transferred from the operator to the operand stack. In other words a tree structure as shown in Fig.3 is formed and stored in the arrays RIGHT[N] and LEFT[N], which contain respectively the right and left operands of the operator represented by N.

The powerful recursion available in Algol 60 enables us to make the section on syntax analysis very brief and compact. Every time an open bracket

is encountered, CREATE is called making available new operator and operand stacks, and when the corresponding closing bracket is encountered these stacks are emptied by transferring their information to the arrays RIGHT and LEFT and the system returns to the previous copy of CREATE, passing back the N-value of the root of the current branch in the global variable ROOT. These roots are represented by boxes in Fig.3.

During the syntax analysis the operators '( ),' disappear and a dummy left leaf is introduced for each unary operator as indicated in Fig.3 by dashed lines.

5.3 Symbolic Algol II Output

At the exit of the first call to procedure CREATE the operator stack PSTACK is empty and only one operand is left in DSTACK; this is the root of the complete tree, namely '/' in Fig.3. It is clear that SA II is obtained by outputting the tree in a specific manner starting from the root. The rule for output may be formulated as follows: at every node, first go down the left branch of the tree until an operand is output, then come back to the node and go down the right branch. This corresponds to the fact that in SA II, it is required that all possible operands or operators on the left should be output before those on the right. Here again, a stack STACK is useful for remembering the node to return to after exhausting a branch of the tree.

Besides the operator and operand names, we need to output opening brackets, commas and closing brackets and this depends on the whole sequence of operators and operands which have been output so far. We therefore make the following rules for output :

Table 3. Output Rules

procedure	sequence	output	enter in TESTST
OPOP	operator preceded by operator	operator (	(
OPDOP	operator preceded by operand	,operator (	,(
OPOPD	operand preceded by operator	operand	
OPDOPD	operand preceded by operand	,operand	,

The integer code representation of '(' or ',' respectively is entered into the first zero element of the array TESTST. In other words TESTST can be considered as a stack for opening brackets and commas. Whenever a closing bracket is output, a pair '(,' must exist at the top. We first remove this

from the stack and then we test if a further closing bracket should be output by examining whether or not another pair '(,' is present. If so, then we output a closing bracket and remove this pair from the stack. At the end of the equation the number of opening and closing brackets should be equal and the stack empty.

During the output process the symbols '+ - * /' are converted to the names SUM, DIFF, MULT, QUOT respectively. Unary + and - are replaced by

SUM(BLANK,   DIFF(BLANK,   (10)

but the dummy operands associated with function operators are suppressed. A positive number represented by

ncc...c   (11)

is replaced by

RNUM(ncc...c)   (12)

where n is a digit 0-9 and cc...c is an arbitrary string of characters not containing any of the reserved symbols

+ - * / , ( ) ; : $ £ & !   .   (13)

This scheme enables a variety of numerical representations such as

$$2 \qquad 3.45 \qquad 6.3 @ 4 \equiv 6.3 \times 10^4 \qquad (14)$$

to be included in the SA I and SA II code but could be generalized if necessary.

6. PROGRAM STRUCTURE

The program is divided into 7 sections ##1 - 7 as shown in Table 4. It has a simple structure consisting of 2 principal blocks, an inner block in which all the processing is done, enclosed within an outer block which is used to set symbolic array dimensions so that these can therefore readily be changed by altering the default settings in #1.2. The variables in the outer block are referred to as global in the usual way, and the statements of the outer block as the 'prelude'. It is convenient to refer to the arrays and procedures of the inner block as 'common' by analogy with Fortran. The main program is a dummy which appears in #7 at the end of the inner block and consists of a single call to procedure MAIN at the head of #3.

It has been found that this type of 'top-down' structure makes the hierarchical organization of Algol 60 programs much clearer, with the leading procedures at the beginning and the subsidiary procedures towards the end. All the system-dependent elements of the program are localized in #6 so that it should only be necessary to modify this one section in order to transfer the program to a different type of computer system.

Table 4

Program Structure

Begin outer block

1. Global variables

   1.1 Declarations
   1.2 Prelude

Begin inner block

2. Common array declarations

3. Principal procedures

   3.1 MAIN
   3.2 PROLOG
   3.3 RESET
   3.4 INPUT
   3.5 CREATE
   3.6 OUTPUT
   3.7 EPILOG

4. Auxiliary procedures

   4.1 Stack manipulation
   4.2 Operator priority
   4.3 Miscellaneous

5. Output procedures

   5.1 Operators and operands
   5.2 Miscellaneous SA II
   5.3 Diagnostic channel NDIAG

6. System-dependent procedures

   6.1 Channels
   6.2 Codes
   6.3 Input channel NIN
   6.4 Main output channel NOUT
   6.5 Diagnostic channel NDIAG

7. Main program

End inner block

STOP:

End outer block

Table 5
List of procedures.

No	Name	Type	Arguments	Scope	Purpose	Mnemonic
				3. Principal procedures		
P3.1	MAIN	P		C	Main control procedure	MAIN control
P3.2	PROLOG	P		C	Initialize the run	PROLOGUE
P3.3	RESET	P		C	Reset variables and arrays for next equation	RESET for next equation
P3.4	INPUT	P	KR	C	Read in the SA I equation	INPUT the equation
P3.4.1	RETURN	BP		INPUT	Condition for returning character to INPUT	RETURN if true
P3.4.2	NEXCHA	IP		INPUT	Fetch the next SA I character	NEXt CHAracter
P3.4.3	RECDOP	P		INPUT	Record an operator + - * / ( ) ,	RECorD OPerator
P3.5	CREATE	P		C	Recursive syntax analyser	CREATE tree
P3.5.1	POPD	IP		CREATE	Fetch operand from DSTACK	POP up operanD
P3.5.2	PUSHD	P	K	CREATE	Store operand K on DSTACK	PUSH down operanD
P3.5.3	POPP	IP		CREATE	Fetch operator from PSTACK	POP up operatoR
P3.5.4	PUSHP	P	K	CREATE	Store operator K on PSTACK	PUSH down operatoR
P3.5.5	FORK	P	KT	CREATE	Set left and right branches of node	define FORK
P3.5.6	FORKB	P	KL,KR	CREATE	Set branches of bracketed node	define FORK (Brackets)
P3.6	OUTPUT	P		C	Output SA II expression	OUTPUT the expression
P3.7	EPILOG	P		C	Close I/O channels	EPILOGue
				4. Auxiliary procedures		
				4.1 Stack manipulation		
P4.1.1	STORE	P	KA,KN,KL	C	Store KN on stack KA and increment KL	STORE on stack
P4.1.2	FETCH	IP	KA,KL	C	Value at top of KA. Decrement KL	FETCH from stack
P4.1.3	POP	IP		C	Value at top of STACK. Decrement MS	POP up
P4.1.4	PUSH	P	KN	C	Store KN on STACK. Increment MS	PUSH down
P4.1.5	PUSHA	P	KN	C	Store KN in ASTORE. Increment MA	PUSH into Astore
P4.1.6	POPT	IP		C	Value at top of TESTST. Decrement MT	POP up from Testst
P4.1.7	PUSHT	P	KN	C	Store KN on TESTST. Increment MT	PUSH into Testst
P4.1.8	LOWERT	P		C	Lower TESTST by 2 levels and clear	LOWER Testst
P4.1.9	PAIR	BP		C	Test for (, at top of TESTST	test for PAIR (,
P4.1.10	PUSHL	P	KN	C	Store KN in LSTORE. Increment ML	PUSH into Lstore
				4.2 Operator priority		
P4.2	PRTY	IP	K	C	Define priority of operator K	operator PRioriTY
				4.3 Miscellaneous		
P4.3	RESETI	P	KA,KD,K		Reset integer array KA of dimension KD to value K	RESET Integer array
				5. Output procedures		
				5.1 Operators and operands		
P5.1.1	OPOP	P	KN	C	Output ⟨operator name⟩(	OPerator-OPerator
P5.1.2	OPDOP	P	KN	C	Output ,⟨operator name⟩(	OPeranD-OPerator
P5.1.3	OPOPD	P	KN	C	Output ⟨operand name⟩	OPerator-OPeranD
P5.1.4	OPDOPD	P	KN	C	Output ,⟨operand name⟩)...)	OPeranD-OPeranD
				5.2 Miscellaneous SA II		
P5.2.1	CHECKL	P	K	C	Count characters and LNFD when required	CHECK Length
P5.2.2	GCHAR	P	K	C	Generate character K	Generate CHARacter
P5.2.3	GWORD	P	KA	C	Generate word or RNUM(number)	Generate WORD
P5.2.4	GENEQU	P		C	Generate 'EQUATE(⟨name⟩'	GENerate EQUate
P5.2.5	GBLANK	P		C	Generate the word BLANK	Generate BLANK
P5.2.6	GRNUM	P		C	Generate 'RNUM('	Generate RNUM
P5.2.7	GENOP	P	K	C	Generate operator name	GENerate OPerator name
				5.3 Diagnostic channel NDIAG		
P5.3.1	PRINTA	P	KN,KA,KD	C	Print name and values of array	PRINT Array
P5.3.2	PRINTC	P	K	C	Print integer code and character K	PRINT Character
				6. System-dependent procedures		
				6.1 Channels		
P6.1.1	CHANLS	P		C	Define I/O channels	CHANnelS
P6.1.2	OPCHAN	P		C	Open I/O channels	OPen CHannels
P6.1.3	OPDIAG	P		C	Open diagnostic channel	OPen DIAGnostics
P6.1.4	CLCHAN	P		C	Close channels	CLose CHANnels
				6.2 Codes		
P6.2	CODES	P		C	Set character codes	set character CODES
				6.3 Input channel NIN		
P6.3	NEXTCH	IP		C	Read next character from channel NIN	NEXT CHaracter
				6.4 Main output channel NOUT		
P6.4.1	CHAR	P	K	C	Output character K on channel NOUT	output CHARacter
P6.4.2	INT	P	K	C	Output integer K on channel NOUT	output INTeger
P6.4.3	LINE	P		C	New line on channel NOUT	new LINE
P6.4.4	TEXT	P	KS	C	Output text on channel NOUT	output TEXT
				6.5 Diagnostic channel NDIAG		
P6.5.1	DCHAR	P	K	C	Output character K on channel NDIAG	output Diagnostic CHARacter
P6.5.2	DINT	P	K	C	Output integer K on channel NDIAG	output Diagnostic INTeger
P6.5.3	DLINE	P		C	New line on channel NDIAG	output Diagnostic LINE
P6.5.4	DTEXT	P	KS	C	Output text on channel NDIAG	output Diagnostic TEXT

Key: B = Boolean, C = Common, I = integer, P = procedure

Table 6

List of variables and arrays.

Name	Type	Dimension	Scope	Purpose	Mnemonic
ASTORE	IA	MAXSTO	C	Stores operand/operator names and symbols	Alpha STORE
BCOUNT	I		G	Counts brackets in SAI input	Bracket COUNT
BLANK	I		G	Integer representation of BLANK	BLANK
CHARCT	I		G	Counts character output	CHARacter CounT
CLOSEB	I		G	Integer representation of ')'	CLOSE Bracket
CLOSED	B		G	Closing bracket encountered	expression CLOSED
COLON	I		G	Integer representation of ':'	COLON
COMA	I		G	Integer representation of ','	COMma
COMMA	B		G	Comma encounted in expression	COMMA found
COPY	B		G	Direct copying in progress	COPY
COPYOF	I		G	Character to switch copying off	COPY OFf
COPYON	I		G	Character to switch copying on	COPY ON
DCOUNT	I		G	Counts diagnostic character output	Diagnostic COUNT
DIAG	B		G	Diagnostics required	DIAGnostics
DIAGON	I		G	Character to switch diagnostics on	DIAGnostics ON
DSTACK	IA	MAXLEV	CREATE	Operand stack	operanD STACK
DUMMY	B		G	Dummy operand encountered	DUMMY operand
DUMOPD	I		G	N-value for dummy operand	DUMmy OPeranD
ENDIN	I		G	Character to end input	END INput
EQUALS	I		G	Integer representation of '='	EQUALS sign
FUNC	IA	MAXS	C	1 if function operator	FUNCtion operator
I	I		INPUT	Current character (NEXCHA, RETURN)	
IA	I		INPUT	Address of previous operator	Address
	I		OPOP	Address of current operator	Address
IC	I		INPUT	Current character (INPUT)	Character
	I		OPOP	(First) character of current operator	Character
	I		GWORD	Current character being output	Character
ID	I		CREATE	Level of stack DSTACK	operanD level
INEXT	I		OUTPUT	Next operator/operand to be output	NEXT
IP	I		INPUT	Previous operator	Previous operator
	I		CREATE	Level of stack PSTACK	oPerator level
IR	I		MAIN	Return marker from input	Return marker
IT	I		FORKB	N-value at top of operator stack	Top
ITOP	I		CREATE	N-value at top of operator stack	TOP
J	I		MAIN	Loop index (equations)	
	I		RESETI	Loop index (array elements)	
	I		OPDOPD	Dummy index for WHILE statement	
	I		GWORD	Loop index (over characters)	
	I		GENEQU	Loop index (over characters)	
	I		PRINTA	Loop index (over array elements)	
K	VI		PUSHD	N-value	
	VI		PUSHP	N-value	
	VI		PRTY	Character	
	VI		RESETI	Value to which KA is reset	
	VI		CHECKL	Length of string to be output	
	VI		GCHAR	Character to be output	
	VI		GENOP	Symbolic representation of operator	
	VI		PRINTC	Character to be printed	
	VI		CHAR	Character to be output	
	VI		INT	Integer to be output	
	VI		DCHAR	Character to be output	
	VI		DINT	Integer to be output	
KA	NIA		STORE	Array to be incremented	Array
	NIA		FETCH	Array to be decremented	Array
	NIA		RESETI	Array to be reset	Array
	VI		GWORD	Address of first character in word	Address
	NIA		PRINTA	Array to be printed	Array
KD	VI		RESETI	Dimension of KA	Dimension
	VI		PRINTA	Dimension of KA	Dimension

KL	VI		FORKB	Left N-value	Left
	NI		STORE	Level of array KA	Level
	NI		FETCH	Level of array KA	Level
KN	VI		STORE	Number to be added to array KA	Number
	VI		PUSH	Number to be added to STACK	Number
	VI		PUSHA	Number to be added to ASTORE	Number
	VI		PUSHT	Number to be added to TESTST	Number
	VI		PUSHL	Number to be added to LSTORE	Number
	VI		OPOP	N-value being output	N-value
	VI		OPDOP	N-value being output	N-value
	VI		OPOPD	N-value being output	N-value
	VI		OPDOPD	N-value being output	N-value
	NS		PRINTA	Name of array	Name
KR	NI		INPUT	Return marker	Return
	VI		FORKB	Right N-value	Right
KS	NS		TEXT	String to be output	String
	NS		DTEXT	String to be output	String
KT	VI		FORK	N-value at top of PSTACK	Top
LEFT	IA	MAXS	C	Value of N at left branch of node	LEFT branch
LINEFD	I		G	Integer representation of line feed	LINEFeeD
LSTORE	IA	MAXLHS	C	Stores name on left-hand-side	Lhs STORE
MØ	I		G	Integer representation of 0	0
M9	I		G	Integer representation of 9	9
MA	I		G	Level of ASTORE	Astore level
MAXEQU	I		G	Maximum number of equations	MAXimum EQUations
MAXLEV	I		G	Maximum level of DSTACK and PSTACK	MAXimum LEVel
MAXLHS	I		G	Maximum length of lhs name	MAXimum LHS
MAXS	I		G		
MAXSTA	I		G	Maximum level of STACK	MAXimum of STAck
MAXSTO	I		G	Maximum contents of ASTORE	MAXimum STore
MINUS	I		G	Integer representation of '-'	MINUS
ML	I		G	Current level of LSTORE	Lstore level
MNUS	IA		C	1 if unary + or -	MiNUS
MS	I		G	Current level of STACK	Stack level
MT	I		G	Current level of TESTST	Testst level
N	I		G	Operand/operator counter	
NDIAG	I		G	Channel for diagnostics	N DIAGnostics
NIN	I		G	Channel for input	N INput
NMAX	I		G	Number of operators/operands	MAXimum
NODE	I		G	Value of N belonging to node	NODE
NOUT	I		G	Channel for output	N OUTput
NUMBER	B		GWORD	Leading digit encountered	Word is a NUMBER
OPENB	I		G	Integer representation of '('	OPEN Bracket
OPIN	I		G	Incoming operator	INcoming OPerator
OPLAST	B		G	Operator last encountered	OPerator LAST
PLUS	I		G	Integer representation for '+'	PLUS
PREFIX	B		G	Unary + or - encountered	operator is a PREFIX
PSTACK	IA	MAXLEV	CREATE	Operator stack	oPerator STACK
RHS	B		G	Right-hand side encountered	Right Hand Side
RIGHT	IA		C	Values of N at right branch of node	RIGHT branch
ROOT	I		G	Root of RHS expression	ROOT
S	IA	MAXS	C	Addresses of names in ASTORE	
SEMI	I		G	Integer representation of ';'	SEMIcolon
SLASH	I		G	Integer representation of '/'	SLASH
STACK	IA	MAXSTA	C	Operator/operand stack	STACK
STAR	I		G	Integer representation of '*'	STAR
TESTST	IA	MAXS	C	Stack for '(,'	TEST STack

Key: A = array, B = Boolean, I = integer, N = call by name, S = string, V = call by value.

Table 5 gives a list of procedures used in the program and Table 6 a list of variables and arrays. Table 7 defines the action of the ICL 4-70 input-output procedures used in #6.

Table 7

ICL System 4 Input-Output Procedures

Name	Type	Arguments	Purpose
CHARIN	IP	KC	Read next character from channel KC
CHAROUT	P	KC,K	Output character K on channel KC
CLOSE	P	KC	Close channel KC
FORMAT	IP	KS	Define integer code for layout string KS
IWRITE	P	KC,KF,K	Output integer K on channel KC with format KF
NEW LINE	P	KC,K	Output K line feeds on channel KC
OPEN	P	KC	Open channel KC
WRITE TEXT	P	KC,KS	Write string KS on channel KC

7. DIAGNOSTICS

Since the SA I input equations are usually fairly straightforward and few in number, as illustrated by Fig.1, errors should be readily detected by eye and no attempt has therefore been made to incorporate the full diagnostics appropriate to a compiler which would make TRANAL unnecessarily complicated. Procedure INPUT performs a limited error analysis to test for overflow of tables LSTORE, ASTORE or an incorrect bracket count: when these occur the program prints a message on channel NOUT and skips to the next equation. Overflow of the other tables is not checked.

The working of the program can be analysed by including a '$' in the input deck, which switches on diagnostic output to channel NDIAG as soon as it is encountered by procedure INPUT. The information that is currently output is indicated in Appendix A, and additional messages can readily be introduced by making use of the diagnostic output procedures in #5.3.

8. CONVERSION TO OTHER COMPUTER SYSTEMS

The following modifications should be all that is needed:

(a) Remove the ICL 4-70 control cards numbered

1-3,659,660,677-700.

(b) Change the hardware representation if necessary, preferably using a context editor.

(c) Rewrite the system-dependent procedures of #6 to take account of local input/output procedures, channel numbers and character codes.

(d) Modify the control symbols $\$$, £, &, ! if required.

(e) Add the control cards appropriate to the local computer system.

Identifiers have been limited to ≤ 6 characters to facilitate the transfer. It may be necessary (as on the ICL 4-70) to remove the serial numbers on the data cards 661-676 before carrying out the Test Run.

ACKNOWLEDGEMENTS

We would like to thank Mr W Holman and Professor P C Poole for helpful discussions.

REFERENCES

[1] K V Roberts and J P Boris. Journ.Comp.Phys. 8, 83 (1971).

[2] M Petravic, G Kuo-Petravic and K V Roberts. Journ.Comp.Phys. 10, 503 (1972).

[3] G Kuo-Petravic, M Petravic and K V Roberts, in 'Cosmic Plasma Physics' Ed. K Schindler, Plenum Press, New York, p.239 (1972).

[4] G Kuo-Petravic, M Petravic and K V Roberts, in 'Computing as a Language of Physics', p.485 IAEA, Vienna (1972); also available as Culham Laboratory Preprint CLM-P270 (Nov.1970).

[5] G Kuo-Petravic, M Petravic and K V Roberts. Culham Laboratory Program Documentation Note CLM-PDN 4/71 (April 1971).

[6] W M Waite. Comm.ACM. 13, 415 (1970).

```
$

NEWRHO=RHO-DT*DIV(RHO*V);

V=(RHO*V+DT*(-GRAD(RHO*TEM+DOT(B,B)/2)
 -DIV(RHO*TEN(V,V)-TEN(B,B))
 +NU*DELSQ(RHO*V)))/NEWRHO;

B=B+DT*(CURL(CROSS(V,B))+ETA*DELSQ(B));

TEM=TEM+DT*(-DIV(TEM*V)+KAPPA*DELSQ(TEM)
    +(2-GAMMA)*SAV(TEM)*DIV(V)
    +(GAMMA-1)*ETA*DOT(CURL(B),CURL(B))/SAV(RHO)
    +(GAMMA-1)*NU*(DOT(CURL(V),CURL(V))+EXP(DIV(V),2)));

£
```

Fig.1 Test Run Input (with diagnostics switched on)

```
EQUATE(NEWRHO,DIFF(RHO,MULT(DT,DIV(MULT(RHO,V)))));

EQUATE(V,QUOT(SUM(MULT(RHO,V),MULT(DT,SUM(DIFF(DIFF(BLANK,GRAD(SUM(
MULT(RHO,TEM),QUOT(DOT(B,B),RNUM(2))))),DIV(DIFF(MULT(RHO,TEN(V,V)),TE
N(B,B)))),MULT(NU,DELSQ(MULT(RHO,V)))))),NEWRHO);

EQUATE(B,SUM(B,MULT(DT,SUM(CURL(CROSS(V,B)),MULT(ETA,DELSQ(B))))));

EQUATE(TEM,SUM(TEM,MULT(DT,SUM(SUM(SUM(SUM(DIFF(BLANK,DIV(MULT(TEM,V))
),MULT(KAPPA,DELSQ(TEM))),MULT(MULT(DIFF(RNUM(2),GAMMA),SAV(TEM)),DIV(
V))),QUOT(MULT(MULT(DIFF(GAMMA,RNUM(1)),ETA),DOT(CURL(B),CURL(B))),SAV
(RHO))),MULT(MULT(DIFF(GAMMA,RNUM(1)),NU),SUM(DOT(CURL(V),CURL(V)),EXP
(DIV(V),RNUM(2))))))));
```

Fig.2 Test Run Output on Channel NOUT.

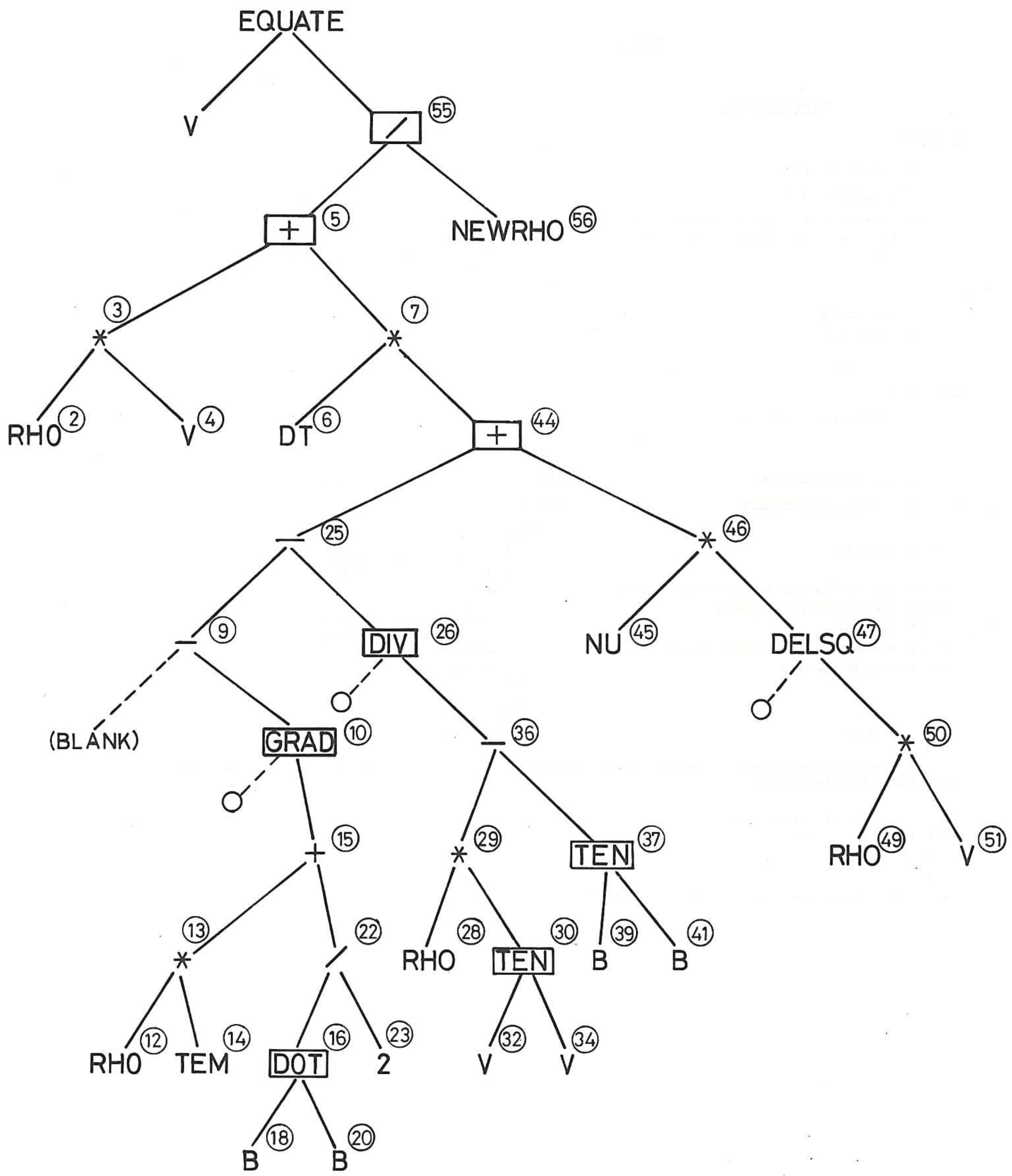


Fig.3 Tree Structure for the Velocity Equation. The numbers in circles indicate N-values. Roots are shown by boxes and dummy operands by ----- 0 or - - - (BLANK).

Program TRANAL

PROGRAM COMMENTARY / ALGOL

OUTER BLOCK

```
begin
```

1. Global variables

```
comment          1.   Global variables;
```

1.1 Declarations

Variables defined at this point are available throughout the program. They include the dimensions of the Common arrays of #2.

```
comment          1.1  Declarations;

  integer  BCOUNT,BLANK,CLOSEB,CHARCT,COLON,COMA,COPYOF,COPYON,
           DCOUNT,DIAGON,DUMOPD,ENDIN,EQUALS,LINEFD,MØ,M9,MA,
           MAXEQU,MAXLEV,MAXLHS,MAXS,MAXSTA,MAXSTO,MINUS,ML,MS,
           MT,N,NDIAG,NIN,NMAX,NODE,NOUT,OPENB,OPIN,PLUS,ROOT,
           SEMI,SLASH,STAR;
  Boolean  CLOSED,COMMA,COPY,DIAG,DUMMY,OPLAST,PREFIX,RHS;
```

1.2 Prelude

Set array dimensions etc.

```
comment          1.2  Prelude;

  MAXEQU:=20;MAXLEV:=20;MAXLHS:=10;
  MAXS:=200; MAXSTA:=50;MAXSTO:=400;
```

INNER BLOCK

```
  begin
```

2. Common array declarations

```
comment          2.   Common array declarations;
    integer array FUNC,LEFT,MNUS,RIGHT,S,TESTST[1:MAXS];
    integer array ASTORE[1:MAXSTO],LSTORE[1:MAXLHS],STACK[1:MAXSTA];
```

3. Principal procedures

```
comment          3.   Principal procedures;
```

P3.1 MAIN Main control procedure

- Initialize the run
- Set or reset variables and arrays for next equation
- Read in and store the SA I statement
- Skip if it is null
- Create the tree by recursive syntax analysis
- Output the SA II statement
- Terminate the run

```
comment P3.1;

    procedure MAIN;
      begin integer IR,J;
        PROLOG;                                  Initialize the run
        for J:=1 step 1 until MAXEQU do
        begin
          RESET;                                 Set or reset variables and arrays for next equation

          INPUT(IR);                             Read in and store the SA I statement
            if IR=1 then go to FINISH;           Skip if it is null
            if IR>1 then go to NEXTEQ;
          CREATE;                                Create the tree by recursive syntax analysis

          OUTPUT;                                Output the SA II statement
    NEXTEQ:
        end;
    FINISH:
        EPILOG;                                  Terminate the run
        go to STOP
      end;
```

P3.2 PROLOG Initialize the run

```
comment P3.2;
    procedure PROLOG;
      begin
        DIAG:=false;                             Diagnostics initially switched off
        CHANLS;OPCHAN;                           Define and open I/O channels
        CODES;                                   Define character codes
        DUMOPD:=MAXS+1                           Set DUMOPD to an N-value that will not be used
      end;
```

P3.3 RESET Reset variables and arrays for next equation

```
comment P3.3;
    procedure RESET;
      begin
        RESETI(FUNC  ,MAXS  ,0); RESETI(LEFT  ,MAXS  ,0);
        RESETI(MNUS  ,MAXS  ,0); RESETI(RIGHT ,MAXS  ,0);
        RESETI(S     ,MAXS  ,0); RESETI(TESTST,MAXS  ,0);
        RESETI(ASTORE,MAXSTO,0); RESETI(LSTORE,MAXLHS,0);
        RESETI(STACK ,MAXSTA,0);

        BCOUNT:=CHARCT:=DCOUNT:=0;
        CLOSED:=COMMA:=COPY:=DUMMY:=false;
        MA:=ML:=MS:=MT:=0;
        N:=1;
        NODE:=OPIN:=ROOT:=0;
        OPLAST:=true;
        PREFIX:=RHS:=false
      end;
```

P3.4 INPUT(KR) Read in the SA I equation

Characters are read by integer procedure NEXCHA

```
comment P3.4;
    procedure INPUT(KR); integer KR;
      begin
        integer I,IA,IC,IP;
```

P3.4.1 RETURN Condition for returning a character to INPUT

If the COPY facility is switched on only the ENDIN character is returned.
If COPY is switched off, blanks and control characters are also ignored.

```
comment P3.4.1;
        Boolean procedure RETURN;
          RETURN:=(I=ENDIN)
            or (I≠BLANK  and I≠LINEFD
            and I≠COPYON and I≠COPYOF
            and I≠DIAGON and not COPY);
```

P3.4.2 NEXCHA Fetch the next SA I character

If the RETURN condition is met the character is returned without being printed. Otherwise various conditions are tested for and markers set accordingly. If diagnostics are required the character and its integer code are printed on channel NDIAG, and if COPY is switched on it is printed on the output channel NOUT, control characters being replaced by blanks.

```
comment P3.4.2;
        integer procedure NEXCHA;
          begin
            for I:=NEXT CH while not RETURN do
            begin
              if I=DIAGON then OPDIAG;
              PRINTC(I);
              if I=COPYON then COPY:=true;
              if I=COPYOF then COPY:=false;
              if I=DIAGON or I=COPYON then I:=BLANK;
              if COPY then GCHAR(I)
            end;
            NEXCHA:=I
          end;
```

P3.4.3 RECDOP Record an operator + - * / ( ) ,

Terminate the previous entry in ASTORE with a semi-colon, enter the operator, record its location and type in array S, note that an operator was last output, and update the operator/operand counter N.

```
comment P3.4.3;
        procedure RECDOP;
          begin
            PUSHA(SEMI);PUSHA(IC);
            S[N]= -MA; OPLAST:=true;
            N:=N+1
          end;
```

Initialize return marker. Blank diagnostic line.

```
        KR:=0; DLINE;
```

Read the SA I characters in turn until a terminating semi-colon or the ENDIN character is reached.

Print the character and its integer code on channel NDIAG if required.

KR≠0 means that an error condition has been encountered.

Otherwise

(a) Store LHS name. Terminate when '=' found. Ignore colon. Record error condition if the name is too long.

```
        for IC:=NEXCHA while IC≠SEMI and IC≠ENDIN do
        begin
          PRINTC(IC); if MA≥MAXSTO-2 then KR:=3;
          if KR=0 then
          begin
            if not RHS then
            begin
              if IC=EQUALS then RHS:=true else
              if IC≠COLON then
              begin if ML ≥ MAXLHS then KR:= 2 else PUSHL(IC) end
            end
              else
```

Analyse RHS:

(b) '+ or -'. Set MNUS to signify a unary operator if this is the first operator encountered, or if it is preceded by '(' or ','. Then record as an operator.

```
            if IC=PLUS or IC=MINUS then
            begin
              IA:= if N>1 then S[N-1] else 0;
              IP:= if IA<0 then ASTORE[-IA] else 0;
              MNUS[N]:= if N=1 or IP=OPENB or IP=COMA then 1 else 0;
              RECDOP
            end
              else
```

(c) '* / ,'. Record as an operator

```
            if IC=STAR or IC=SLASH or IC=COMA then RECDOP
              else
```

(d) '('. Increment the bracket count, and change any preceding operand to a function operator. Then record '(' as an operator.

```
            if IC=OPENB then
            begin
              BCOUNT:=BCOUNT+1;
              IA:= if N>1 then S[N-1] else 0;
              if IA>0 then begin S[N-1]:= -IA; FUNC[N-1]:=1 end;
              RECDOP
            end
              else
```

(e) ')'. Decrement the bracket count, and record ')' as an operator.

```
            if IC=CLOSEB then begin BCOUNT:=BCOUNT-1;RECDOP end
              else
```

(f) Any other symbol. Store in ASTORE, and if it was preceded by an operator record the first character position of the current string and increment the operand/operator counter. Note that an operand is being processed.

```
        begin
          PUSHA(IC);
          if OPLAST then begin S[N]:=MA; NA:=N+1 end;
          OPLAST:= false
        end
      end
    end;
```

Print the final character which will be ';' or ENDIN. Note the total number of operators and operands to be processed.

```
    PRINTC(IC);
    NMAX:=N; N:=0;
```

Store the terminating semi-colon, and print diagnostics if required.

```
    if IC:=SEMI then
    begin
      PUSHA(SEMI); DLINE;
      PRINTA('ASTORE',ASTORE,MA  );
      PRINTA('S     ',S     ,NMAX);
      PRINTA('FUNC  ',FUNC  ,NMAX);
      PRINTA('MNUS  ',MNUS  ,NMAX)
    end;
```

Set return markers and print failure messages.

```
    if IC=ENDIN then KR:=1;
    if KR=2 then TEXT('LHS-OVERFLOW');
    if KR=3 then TEXT('RHS-OVERFLOW');
    if BCOUNT≠0 then
    begin
      KR:=4;
      TEXT('BRACKET-COUNT'); INT(BCOUNT)
    end;
    if KR>1 then LINE
  end;
```

P3.5 CREATE Create the tree by recursive syntax analysis

```
comment P3.5;
    procedure CREATE;
      begin
        integer ID,IP,ITOP;
        integer array DSTACK,PSTACK[1:MAXLEV];
```

There is one copy of the operand and operator stacks for each level at which CREATE is called, and the stack manipulation procedures work at each individual level as follows:

Stack manipulation procedures

```
comment stack manipulation procedures;
```

P3.5.1 POPD, P3.5.2 PUSHD(K) } Control the operand stack DSTACK

```
        integer procedure POPD;POPD:=FETCH(DSTACK,ID);
        procedure PUSHD(K);value K;integer K;STORE(DSTACK,K,ID);
```

P3.5.3 POPP, P3.5.4 PUSHP(K) } Control the operator stack PSTACK

```
        integer procedure POPP;POPP:=FETCH(PSTACK,IP);
        procedure PUSHP(K);value K;integer K;STORE(PSTACK,K,IP);
```

P3.5.5 FORK(KT)

Set the right and left branches for nodes without brackets

```
        procedure FORK(KT);value KT;integer KT;
          begin
            RIGHT[KT]:=POPD;
            LEFT[KT]:= if MNUS[KT]=1 then DUMOPD else POPD;
            PUSHD(KT)
          end;
```

P3.5.6 FORKB(KL,KR)

Set the right and left branches for nodes with brackets

```
        procedure FORKB(KL,KR);value KL,KR;integer KL,KR;
          begin integer IT;
            IT:=POPP;
            RIGHT[IT]:=KR; LEFT[IT]:=KL;
            PUSHD(IT); COMMA:=false
          end;
```

Initialization

Zero the operand and operator stacks.
Clear the markers for bracket closure and comma.

```
comment initialisation
        ID:=IP:=0;
        RESETI(DSTACK,MAXLEV,0);RESETI(PSTACK,MAXLEV,0);
        CLOSED:=COMMA:=false;
```

Syntax analysis

Scan over all operators and operands until a closing bracket is found or the end of the stack is reached.

If the Nth item is an operand (S > 0) then enter it on the operand stack and continue. Otherwise check whether the incoming operator OPIN is:

(a) Opening bracket. Enter a new copy of CREATE, and analyse until the corresponding closing bracket is encountered; then return and reset the marker. COMMA=true means that a comma has been encountered in the top-most copy of CREATE from which we have just returned. For example, take the expression

...CROSS(V,B)+...

The topmost copy of CREATE contained the comma as its root and V and B as its left and right branches respectively. While in that copy we made COMMA=true and NODE, a global variable, the value of N corresponding to the operator comma. When we return to the previous copy of CREATE, i.e. that which contains the operator CROSS, we transfer the left and right branches of NODE to that of CROSS which is represented by ITOP, the operator which immediately preceded the opening bracket and is at the top of the operand stack.

If no comma has been encountered we examine ITOP:

(i) Operator stack empty, or ITOP not a function operator. Push the root of the expression inside the pair of brackets from which we have returned on to the operand stack.

(ii) Unary function operator. Make the left leaf of ITOP a dummy operand, and the right leaf equal to the root of the bracketed expression.

(b) Closing bracket. Examine one by one each entry ITOP that is currently at the top of the operator stack. The procedure FORK sets the right branch of ITOP to the entry at the top of the operand stack, and the left branch either to the next entry (binary operator) or to a dummy operand (unary operator). Then ITOP is placed on the operand stack.

A comma is recorded if one has been found.

Allow for a final increment of N in the 'for' statement before returning.

When the operator stack is empty then the root NODE of the bracketed expression is given by the entry at the top of the operand stack. Print value of NODE if diagnostics required.

(c) Any other operator. Examine one by one each entry ITOP that is currently at the top of the operator stack, until either the stack is exhausted, or the priority of ITOP is less than that of the incoming operator. Set the FORK as in (b).

If an entry ITOP remains return it to the stack, and then place N on the stack.

At the exit from the outermost copy of CREATE, the last operator or operand has been encountered and we have reached the end of the statement. Unload the operator and operand stacks setting the right and left branches. The last operator (which has been placed on the operand stack by FORK) is the root.

Print diagnostics if required.

```
comment syntax analysis

    for N:N+1 while not CLOSED and N<NMAX do
    begin
      if S[N]>0 then PUSHD(N)
        else
      begin
        OPIN:=ASTORE[-S[N]];
        if OPIN=OPENB then
        begin
          CREATE; CLOSED:=false;

          if COMMA then FORKB(LEFT[NODE],RIGHT[NODE])

            else
          begin
            ITOP:=POPP; if ITOP≠0 then PUSHP(ITOP);
            if ITOP=0 or FUNC[ITOP]=0 then PUSHD(NODE)

            else FORKB(DUMOPD,NODE)
          end
        end
          else
        if OPIN=CLOSEB then
        begin
          for ITOP:=POPP while ITOP>0 do
          begin
            FORK(ITOP);
            if ASTORE[-S[ITOP]]=COMA then COMMA:= true
          end
          N:=N-1;
          NODE:=POPD; CLOSED:= true;
          DLINE;DTEXT('NODE');DINT(NODE)
        end
          else
        begin
          for ITOP:=POPP while ITOP>0
            and PRTY(ASTORE[-S[ITOP]]≥ PRTY(OPIN)
            do FORK(ITOP);
          if ITOP>0 then PUSHP(ITOP);
          PUSHP(N)
        end
      end
    end;
  if not CLOSED and N=NMAX then
  begin
    for ITOP:=POPP while ITOP>0 do FORK(ITOP);
    ROOT:=POPD;
    DLINE;DTEXT('ROOT');DINT(ROOT);DLINE;
    PRINTA('RIGHT',RIGHT,NMAX);
    PRINTA('LEFT ',LEFT ,NMAX);
    DLINE
  end
end;
```

P3.6 OUTPUT Output the SA II statement

Initialize counters and marker.
Output 'EQUATE(<name>'.
Record '(' and count <name> as a left operand.
Set the root of the right-hand side expression obtained from CREATE at the bottom of the array STACK, and output it as the second operand of EQUATE.
Examine at each stage the top item on STACK until the latter is empty, and

(a) Choose left or right branch. The variable INEXT is set to the N-value for the left branch if an operator has last been output, and in this case the Boolean variable PREFIX is set to true if this operator is a unary + or -. If an operand has last been output then it is set to the N-value for the right branch.

(b) Output the operator or operand name. If it is an operator name, place it on the stack, then proceed as follows:

Previous name	Current name	Output
operator	operator	<name>(
operand	operator	,<name>(
operator	operand	<name>
operand	operand	,<name>))...)

(For details see P5.1.1 - 5.1.4)

When the stack is empty output a semi-colon and terminate the line.

```
comment P3.6;
  procedure OUTPUT;
    begin integer INEXT
      MS:=MT:=0; DUMMY:= false;
      GENEQU;
      PUSHT(OPENB);
      PUSH(ROOT); OPDOP(ROOT);
      begin if OPLAST then
        begin
          INEXT:=LEFT[N];PUSH(N);
          PREFIX:=MNUS[N]=1
        end else
        INEXT:=RIGHT[N];

        if S[INEXT]<0 then
        begin PUSH(INEXT);
          if OPLAST then OPOP(INEXT) else OPDOP(INEXT)
            end else
        if OPLAST then OPOPD(INEXT) else OPDOPD(INEXT)
      end;

      GCHAR(SEMI);LINE
    end;
```

P3.7 EPILOG Close I/O channels

```
comment P3.7;
  procedure EPILOG; CLCHAN;
```

---

4. Auxiliary procedures

4.1 Stack manipulation

```
comment          4.1   Stack manipulation
```

```
comment P4.1.1 - P4.1.10;
```

P4.1.1 STORE(KA,KN,KL) Store KN on stack KA and increment level KL

```
  procedure STORE(KA,KN,KL); value KN;
    integer array KA; integer KN,KL;
    begin KL:=KL+1;KA[KL]:=KN end;
```

P4.1.2 FETCH(KA,KL) Value at top of STACK. Decrement KL

If the stack is empty then zero is returned and KL is not decremented.

```
    integer procedure FETCH(KA,KL); integer array KA; integer KL;
    begin
      FETCH:= if KL=0 then 0 else KA[KL];
      if KL≠0 then begin KA[KL]:=0; KL:=KL-1 end
    end;
```

P4.1.3 POP Value at top of STACK. Decrement MS

```
  integer procedure POP; POP:=FETCH(STACK,MS);
```

P4.1.4 PUSH(KN) Store KN on STACK. Increment MS

```
  procedure PUSH(KN);value KN;integer KN;STORE(STACK,KN,MS);
```

P4.1.5 PUSHA(KN) Store KN in ASTORE. Increment MA

```
  procedure PUSHA(KN);value KN;integer KN;STORE(ASTORE,KN,MA);
```

P4.1.6 POPT Value at top of TESTST. Decrement MT

```
  integer procedure POPT;POPT:=FETCH(TESTST,MT);
```

P4.1.7 PUSHT(KN) Store KN in TESTST. Increment MT

```
  procedure PUSHT(KN);value KN;integer KN;STORE(TESTST,KN,MT);
```

P4.1.8 LOWERT Lower TESTST by 2 levels and clear

```
  procedure LOWERT;
    begin TESTST[MT]:=TESTST[MT-1]:=0;MT:=MT-2 end;
```

P4.1.9 PAIR Test for '(,' at top of TESTST

```
  Boolean procedure PAIR;
    PAIR:=TESTST[MT]=COMA and TESTST[MT-1]=OPENB;
```

P4.1.10 PUSHL(KN) Store KN in LSTORE. Increment ML

```
  Procedure PUSHL(KN);value KN;integer KN;STORE(LSTORE,KN,ML);
```

4.2 Operator priority

P4.2 PRTY(K) Define priority of operator K

```
comment             4.2   Operator priority;

comment P4.2;
    integer procedure PRTY(K); value K;integer K;
      PRTY:= if K=COMA then 1
        else if K=PLUS or K=MINUS then 2
        else if K=STAR or K=SLASH then 3
        else 10;
```

4.3 Miscellaneous

P4.3 RESETI(KA,KD,K) Reset integer array KA of dimension KD to value K

```
comment             4.3   Miscellaneous;

comment P4.3;
    procedure RESETI(KA,KD,K);value KD,K;integer array KA;
      integer KD,K;
      begin integer J;
        for J:=1 step 1 until KD do KA[J]:=K
      end;
```

---

5. Output procedures

5.1 Operators and operands

```
comment             5.   Output procedures;

comment             5.1   Operators and operands
```

P5.1.1 OPOP(KN) Output an operator which has been preceded by an operator

If the operator is one of + - * / then output operator or if it is a function operator output its name.

Output (and record in the TESTST stack that this has been done.

Note that the name last output is that of an operator.

```
comment P5.1.1;
    procedure OPOP(KN);value KN;integer KN;
      begin integer IA, IC;
        IA:= -S[KN]; IC:=ASTORE[IA];
        if FUNC[KN]=0 then GENOP(IC) else GWORD(IA);
        GCHAR(OPENB);PUSHT(OPENB);
        OPLAST:=true
      end;
```

P5.1.2 OPDOP(KN) Output an operator which has been preceded by an operand

Output a comma unless the previous operand was a dummy, in which case the market is reset. In either case record in the TESTST stack that a (real or fictitious) comma has been output. Then proceed as in P5.1.1.

```
comment P5.1.2;
    procedure OPDOP(KN);value KN;integer KN;
      begin if DUMMY then DUMMY:= false else GCHAR(COMA);
        PUSHT(COMA);OPOP(KN)
      end;
```

P5.1.3 OPOPD(KN) Output an operand which has been preceded by an operator

If the operand is a dummy then either output the word BLANK in case the operator was + or - or set the DUMMY marker; otherwise output the operand name.

Note that the name last output is that of an operand.

```
comment P5.1.3;
    procedure OPOPD(KN);value KN;integer KN;
      begin if KN=DUMOPD;
        then begin if PREFIX then GBLANK else DUMMY:=true end
        else GWORD(S[KN]);
        OPLAST:=false
      end;
```

P5.1.4 OPDOPD(KN) Output an operand which has been preceded by an operand

J is a dummy index required by Algol 60.
Output a comma unless the previous operand was a dummy, in which case the marker is reset. Record a comma in the TESTST stack. Output the operand name and note its type.

Output a series of closing brackets, one for each pair '(', at the top of the stack, lowering the level by 2 each time.

```
comment P5.1.4;
    procedure OPDOPD(KN);value KN;integer KN;
      begin integer J;
        if DUMMY then DUMMY:=false else GCHAR(COMA);
        PUSHT(COMA);
        GWORD(S[KN]);OPLAST:=false;

        for J:=0 while MT≥2 and PAIR do
          begin GCHAR(CLOSEB);LOWERT end
      end;
```

5.2 Miscellaneous SA II

```
comment             5.2   Miscellaneous SA II;
```

P5.2.1 CHECKL(K) Count characters and issue LINEFEED when required

CHARCT is set to the position of the last of the K characters to be output.
70 characters are allowed on each line.

```
comment P5.2.1;
    procedure CHECKL(K);value K;integer K;
      begin if CHARCT>70-K then begin LINE;CHARCT:=0 end;
        CHARCT:=CHARCT+K
      end;
```

P5.2.2 GCHAR(K) Generate character K

```
comment P5.2.2
    procedure GCHAR(K);value K;integer K;begin CHECKL(1);CHAR(K)end;
```

P5.2.3 GWORD(KA) Generate word or RNUM(number) defined by location KA of ASTORE

If the first character of the word is a digit 0-9, output the string 'RNUM('.

Output the word up to but not including the terminating semicolon.

For a number, output a closing bracket.

```
comment P5.2.3;

  procedure GWORD(KA);value KA;integer KA;
    begin Boolean NUMBER;integer IC,J;
      IC:=ASTORE[KA];J:=0;
      NUMBER:=IC ≥ MØ and IC ≤ M9;
      if NUMBER then GRNUM;
      for J:=J+1 while IC≠SEMI do
        begin GCHAR(IC);IC:=ASTORE[KA+J] end;
      if NUMBER then GCHAR(CLOSEB)
    end;
```

P5.2.4 GENEQU Generate 'EQUATE(⟨name⟩'

```
comment P5.2.4;

  procedure GENEQU;
    begin integer J;
      LINE;CHARCT:=7;TEXT('EQUATE(');
      for J:=1 step 1 until ML do GCHAR(LSTORE[J])
    end;
```

P5.2.5 GBLANK Generate the word 'BLANK'

```
comment P5.2.5;

  procedure GBLANK;begin CHECKL(5);TEXT('BLANK')end;
```

P5.2.6 GRNUM Generate 'RNUM('

```
comment P5.2.6;

  procedure GRNUM;begin CHECKL(5);TEXT('RNUM(')end;
```

P5.2.7 GENOP(K) Generate name of operator +,-,*,/

```
comment P5.2.7;

  procedure GENOP(K);value K;integer K;
    begin if K=PLUS then begin CHECKL(3);TEXT('SUM')end
      else if K=MINUS then begin CHECKL(4);TEXT('DIFF')end
      else if K=STAR  then begin CHECKL(4);TEXT('MULT')end
      else if K=SLASH then begin CHECKL(4);TEXT('QUOT')end
    end;
```

5.3 Diagnostic channel NDIAG

```
comment              5.3    Diagnostic channel NDIAG;
```

P5.3.1 PRINTA(KN,KA,KD) Print name KN and values of array KA with dimension KD

10 values are allowed on each line

```
comment P5.3.1;

  procedure PRINTA(KN,KA,KD);value KD;string KN; integer array KA;
    integer KD;
    begin integer J;
      DLINE;DTEXT(KN);DLINE;
      for J:=1 step 1 until KD do
      begin
        DCHAR(BLANK);DINT(KA[J]);
        if 10*(J÷10)=J then DLINE
      end;
      DLINE
    end;
```

P5.3.2 PRINTC(K) Print integer code followed by character

7 values are allowed on each line

```
comment P5.3.2;

  procedure PRINTC(K);value K;integer K;
    begin
      if DCOUNT ≥ 7 then begin DLINE;DCOUNT:=0 end;
      DCHAR(BLANK);DCHAR(K);DCHAR(BLANK);DINT(K);
      DCOUNT:=DCOUNT+1
    end;
```

---

6. System-dependent procedures

```
comment              6.    System-dependent procedures;
```

The implementation of the following procedures is specific to the ICL System 4, although their names and calling sequences are standard. To transfer TRANAL to another type of computer, implement a corresponding set of procedures using local I/O facilities.

6.1 Channels

```
comment              6.1    Channels;
```

P6.1.1 CHANLS Define channels for input,output, diagnostics

```
comment P6.1.1 - P6.1.4

  procedure CHANLS;begin NIN:=210;NOUT:=200;NDIAG:=201 end;
```

P6.1.2 OPCHAN Open I/O channels

```
  procedure OPCHAN;begin OPEN(NIN);OPEN(NOUT) end;
```

P6.1.3 OPDIAG Open diagnostic channel

```
procedure OPDIAG;
  begin
    if not DIAG and NDIAG≠NOUT then OPEN(NDIAG);
    DIAG:=true
  end;
```

P6.1.4 CLCHAN Close channels

```
procedure CLCHAN;
  begin
    CLOSE(NIN);CLOSE(NOUT);
    if DIAG and NDIAG≠NOUT then CLOSE(NDIAG)
  end;
```

6.2 Codes

P6.2 CODES Set character codes

```
comment               6.2    Codes;

comment P6.2;

  procedure CODES;
    begin
      BLANK := 64; CLOSEB:= 93; COLON := 122; COMA  := 107;
      COPYON:= 80; COPYOF:= 90; DIAGON:= 91;  ENDIN := 74;
      EQUALS:= 126; LINEFD:= 21; MØ    := 240; M9    := 249;
      MINUS := 96; OPENB := 77; PLUS  := 78;  SEMI  := 94;
      SLASH := 97; STAR  := 92
    end;
```

6.3 Input channel NIN

P6.3 NEXTCH Fetch next character

```
comment               6.3    Input channel NIN;

comment P6.3;
  integer procedure NEXTCH;NEXTCH:=CHARIN(NIN);
```

6.4 Main output channel NOUT

P6.4.1 CHAR(K) Output character K as channel NOUT

P6.4.2 INT(K) Output integer K on channel NOUT

P6.4.3 LINE New line on channel NOUT

P6.4.4 TEXT(KS) Output text on channel NOUT

```
comment               6.4    Main output channel NOUT;

comment P6.4.1 - 6.4.4;
  procedure CHAR(K);value K;integer K;CHAROUT(NOUT,K);

  procedure INT(K);value K;integer K;IWRITE(NOUT,FORMAT('-NDDD'),K);

  procedure LINE;NEW LINE(NOUT,1);

  procedure TEXT(KS);string KS;WRITE TEXT(NOUT,KS);
```

6.5 Diagnostic channel NDIAG

P6.5.1 DCHAR(K) Output character on channel NDIAG

P6.5.2 DINT(K) Output integer on channel NDIAG

P6.5.3 DLINE New line on channel NDIAG

P6.5.4 DTEXT(KS) Output text on channel NDIAG

```
comment               6.5    Diagnostic channel NDIAG;

comment P6.5.1 - P6.5.4;

  procedure DCHAR(K);value K;integer K;
    if DIAG then CHAROUT(NDIAG,K);

  procedure DINT(K);value K;integer K;
    if DIAG then IWRITE(NDIAG,FORMAT('-NDDD'),K);

  procedure DLINE;if DIAG then NEW LINE(NDIAG,1);

  procedure DTEXT(KS);string KS;if DIAG then WRITE TEXT(NDIAG,KS);
```

---

7. Main program

The main program of the inner block consists of a single call to procedure MAIN. This enables the program to be arranged in hierarchically descending order. The run is terminated by 'go to STOP'.

```
comment            7.    Main program;

    MAIN
  end;
 STOP:
end
```

CULHAM LABORATORY
LIBRARY
- 4 MAY 1976