D.R. TAYLOR 19A.

INTERNAL USE ONLY

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED .

SCIENTIFIC PROGRAMMING DEPT.
I.C.T. 1900 SERIES

Fortran Note 14
24.9.65

Some Aspects of Expression Evaluation in the FORTRAN IV Compiler

This document attempts to show how expressions in FORTRAN are analyzed. The first part (pages 1-10) introduces the transformation of expressions from standard to Polish notation. This is followed by sections (pages 11-31) which indicate in general terms how the Polish notation equivalent of the original expression is used to define operations which can be separately compiled. This is followed (pages 31-41) by more detailed descriptions of how the operations are represented within the compiler. There follows a description (pages 42-61) of how the actual instructions for these operations are compiled. The document finishes with a short description of the compiler's internal representation of the various items of an expression.

TABLE OF CONTENTS

The Evaluation of Expressions in FORTRAN 1

Generation of PN equivalent for a FORTRAN expression 4

Rules for Generating PN List 4
Unary Algebraic Operations 6
Function and Array References 7
Commas within Expressions 7

Range of Operators 8

Partial PN List - The Operation List 9

The Actual Evaluation of Operations in FORTRAN 11

'Store Auxiliary' Operations 12
Operation Mode - Mode Auxiliary Operation 16

Temporary Store Allocation 20
Operands 27
Relationship between Operators and Operands 27
Summary of Operator Association 30

Practical Compilation 31

Operation Compilation (I) 32

1. Binary Operations 32
1.1 Arithmetic Binary Operations 32
1.1.1 Standard Binary 33
1.1.2 Double Precision Binary 34
1.1.3 Complex Binary 34
1.1.4 Small Integer Binary 35
1.1.5 Standard Exponentiation 35
1.1.6 Small Integer Exponentiation 36
1.2 Logical Binary Operations 36
2. Relational Binary Operation 37
3. Unary Minus 38
4. LOGICAL .NOT. 38
5. Assignment Operations 38
6. Store Auxiliary Operations 40
6.1 Load Acc. 40
6.2 Store Acc. 41
6.3 Load X3 41
6.4 Store X3 41

Group Identifiers 42

Instruction Generator (I) 43

Group Table 43
Position Tables 44
Instruction Tables and Relativisor Tables 47
(Instruction Generator Examples) 48

TABLE OF COMMENTS (Continued)

Restriction on the General Use of the Instruction Generator 55

Instruction Generator (II) 56

Differences between AC0M and AC02 entries to Instruction Generator 56

Operation Compilation (II) 58
- Function Operations 58
- Intrinsic Functions 59
- Array Operations 59
- Function (Array) Argument (Subscript) Lists 60

Construction of the PN List in the Compiler 62

- PN List Item 62
- Complex Constants 65
- Standard and Intrinsic Functions 66
- Operator List 66

THE EVALUATION OF EXPRESSIONS IN ~~FORTRAN IV~~ PSycle

General A ~~FORTRAN~~ Psycle expression is a sequence of operands, operators and ~~separators~~ bracket followed by a terminator, which in itself constitutes an algebraic, relational or logical expression.

An operand is any constant or any name (e.g. A, JACK, 3·5)

An operator is a symbol from the set

+,-,*,/,**	(algebraic)
.EQ.,.NE.,.LE.,.GE.,.LT.,.GT.	(relational)
.AND.,.OR.,.NOT.	(logical)
=	(assignment)

A ~~Separator~~ bracket is a symbol from the set ( ), [,]

A Terminator is the meta-symbol 'E of S'

Under certain circumstances ',' may act as a terminator.

To evaluate any expression it is necessary to determine the order of operations implied in that expression.
For example for the expression

A+B*C

it is necessary to know that B*C is to be evaluated before A is added to the result. On the other hand, for the expression

(A+B)*C

the quantity A+B is to be evaluated and then the result is to be multiplied by C.

For an expression which contains no brackets, the effective order of operations is determined by the relative 'Weights' of the various operators. An operation involving an operator of greater weight will take place before an operation involving an operator of lesser weight if both operations share a common operand. For two operations involving operators of the same weight, where, both operations share a common operand, the left most operation will be performed first.

*(multiply) is defined to have a greater weight than '+' (add).

In the following expression

A + B*C + D*E + F

The order of operations is

1. B*C
2. Result + A
3. D*E
4. Result of 3 + Result of 2
5. Result of 4 + F

For an expression which contain brackets, the effective order of operations is determined as before with the following rule: All operations within brackets are performed prior to any operation outside the brackets which requires the result of the bracketed operation as an operand.

Thus in the expression

A + B*C + (D*E + F)

The order of operations is

1. B*C
2. Result + A
3. D*E
4. Result of 3 + F
5. Result of 4 + Result of 2

Note that A+B*C could be evaluated before D*E+F was looked at, because the result of the bracketed operations was not required for those operations.

To determine the order of operations, it is necessary to scan the expression backwards and forwards to compare operator weights and bracket levels. Within the Compiler this is not an easy or efficient operation either in space or time.

Fortunately the expressions can be fairly easily manipulated within the compiler so that the operations are defined sequentially. In other words, in the manipulated expression the operators are found in the order in which they are required.

For example, the expression

A + B * C + D*E + F

is converted to

ABC* + DE * + F +

and the expression

A + B* C + (D*E+F)

is converted to

ABC* + DE*F + +

Note that in the second case, the brackets have been eliminated from the expression.

The revised expression is in 'post fix' notation. That is at the time of an operation, the operators immediately follow the operands with which they are associated .

To demonstrate this, consider the ~~expression~~ following converted expression

ABC* + DE* + F +

If at each Stage in evaluating this expression, the result of an operation (R) replaces the operator and operands used in that operation, the following phases occur during the evaluation.

Phase	Post Fix Operation	Resultant Expression	Complete Operation
1.	BC*	A R_1 + DE* + F +	B*C
2.	AR_1 +	R_2 DE* + F +	B*C + A
3.	DE*	R_2R_3 + F +	D*E
4.	R_2R_3+	R_4F +	D*E + B*C + A
5.	R_4F +	R_5	D*E + B*C + A + F

This Particular Post Fix notation equivalent to the original ~~FORTRAN~~ expression is called the 'Polish' Notation (PN) equivalent, ~~and~~ it has the characteristics that the operators all appear in order of use, and that no ~~brackets~~ parentheses or bracket appear in the original ~~FORTRAN~~ expression ~~pass over to~~ in the Polish equivalent.

The Generation of the PN Equivalent of a ~~FORTRAN~~ Expression.

To form the PN Equivalent, the original ~~FORTRAN~~ expression is written element by element into two Lists - the Operand List (PN List) and the Operator List - both initially empty, according to the set of rules given in this section.

The relative 'weights' of the operators for ~~FORTRAN~~ expressions is given in the following table.

1	←	Assignment
2	.OR.	Logical
3	.AND.	Logical
4	.NOT.	Logical
5	.EQ., .NE., .GT., .LE., .LT., .GE.	Relational
6	+, -	Algebraic
7	*, /	Algebraic
8	~~**~~ ↑	Algebraic

The numerical 'weights' are meaningless in themselves. However, an operation involving an operator of greater weight will be evaluated before an operation involving an operator of lesser weight if the two operations share a common operand.

The 'top item' of a list is the last item put in the list or currently in the list.

The rules for generating the PN list are set out below

1. Elements are extracted from the original ~~FORTRAN~~ expression in their original order.
2. If the element is an operand it is inserted at the top of the operand list.
3. If the element (e) is an operator, it is compared for weight with the top item (t) in the operator list.
   - 3.1. If 'e' is greater than 't', then 'e' becomes the top item in the operator list.

an 'opening bracket' is '(' or '['

3.2. If 't' is ~~'('~~, then 'e' becomes the top item in the operator list.

3.3. If the operator list is empty, then 'e' becomes the top item in the operator list.

3.4. If 'e' is not greater than 't' and if 3.2 or 3.3 does not apply then 't' is transferred to the top of the operand list. The element is then compared with the new 't' according to the rules 3.1, 3.2, 3.3 and 3.4.

an opening bracket

4. If the element is ~~the separator '('~~ it is inserted at the top of the operator list.

a closing bracket (i.e. ')' or ']', an opening bracket

5.1 If the element is ~~')'~~ then unless 't' is ~~'('~~ or the operator list is empty, 't' is transferred to the top of the operand list and the element is then compared with the new 't' according to this rule.

5.2 If 't' is ~~'('~~, it is removed from the operator list and the comparison stops.

5.3 If there are no items in the operator list the comparison stops. ')' never goes into either list.

New paragraph

6. If the element is ',' (as a separator) it acts as an operator of the same weight as '='.

7. If the element is 'E of S' it acts in the same manner as ')' except that the comparison stops only when there are no items in the operator list.

As an example of the generation of a PN equivalent, consider the expression

A + B * (C - D)

This will eventually be converted into

ABCD - * +

by the following steps.

Step	Operation	Rule	List Contents Operand (PNL)	Operation (OL)
1	A → PNL	2	A	Empty
2	'+' → OL	3.3	A	+
3	B → PNL	2	AB	+
4	'*' → OL	3.1	AB	+*
5	'(' → OL	4	AB	+*(
6	C → PNL	2	ABC	+*(
7	'-' → OL	3.2	ABC	+*(-
8	D → PNL	2	ABCD	+*(-
9	')' compared with OL			
	'-' → PNL	5.1	ABCD -	+*(
	'(' removed from OL	5.2	ABCD -	+*
10	'E of S' compared with OL			
	'*' → PNL	7	ABCD - *	+
	'+' → PNL	7	ABCD - *+	Empty

The final PN Equivalent for the original FORTRAN expression is the final Operand List.

The Unary Algebraic Operations (-A, +A)

The symbol '-' may stand for two different operations, one involving two operands (eg A-B) and one involving one operand (eg - A).

Although these operations can be differentiated with the expression in standard form, this is not true with the expression in Post Fix notation. For example, the expression

X + A *(-B)

would turn into

X AB - *+

As it stands, this is not distinguishable from the PN equivalent for The expression.

+ X * (A-B)

where '+' is an operation involving one operand (unary +)

To avoid ambiguity with these two symbols (+ and -), when the PN equivalent for an operation using '-' as a unary operation is generated, the operator is converted to a new operator '$\underset{u}{-}$' before the equivalent is made. This operator has the same weight as '-'.

Thus the PN equivalent for X+A* (-B) is XAB $\underset{u}{-}$ *+

The operator '+' as a unary operator is always redundant and is ignored when making PN equivalents.

Thus the PN equivalent for +X*(A-B) is XAB - *

Function and Array References

Not all FORTRAN expressions involve only simple elements. For example the expression A + SIN (X) contains the complex element SIN(X) as an operand of the '+' operator.

In order to accommodate elements of this type it is necessary to extend the rules and definitions described so far.

In an expression, A complex element of the form Name (- - -) is conceptually converted to a series of simple elements of the form

Name op (-----)

before the PN equivalent for the expression is found.

'Name' is the name of a Function or of an Array op is the pseudo-operator f or a depending on whether the complex element is a Function reference (f) or an array reference (a)

For example, the expression A + SIN (X) is conceptually converted to the form A + SIN f (X) before any post-fix notation equivalent is formed.

The operators f and a are of the same weight, and are of greater weight than **.

Thus the PN equivalent form for A + SIN (X) is

A SIN X f +

If B is an array name, then the PN equivalent form for B(I) + SIN(X) will be

BI a SIN X f +

Commas Within Expressions

Commas may occur in three places within FORTRAN expressions (including assignment statements) as follows.

1. As a separator on the l.h.s. of an assignment statement
   eg. A,B,C = X
   The PN equivalent of this statement is
   AB,C, X=
2. As an argument separator in a function reference
   eg FCN (X,Y,Z)
   The PN equivalent of this function reference is
   FCN XY, Z, f
3. As a subscript separator in an array reference
   eg ARRAY (I,J,K)
   The PN equivalent of this array reference is
   ARRAY IJ,K, a

The 'Range' of Operators

In a normal FORTRAN expression the 'range' of an operator is intuitively obvious. Thus in the expression A+B*COS(C), the range of the + is the operand A and the result of the operation B*COS(C). The range of * is the operand B and the result of the operation COS(C). The range of the pseudo-operator f is the name COS and the single argument C.

The same expression in PN form is AB COS C f *+

Breaking this down into operations, replacing an operation in the list by its result,

0. AB COS C f * +
1. AB R_1*+ R_1 = COS(C)
2. AR_2+ R_2 = B*COS(C)
3. R_3 R_3 = A+B*COS(C)

it can be seen that at the time of a particular operation, the operands for that operation are in standard positions relative to the operator. The positions of these operands determine the 'Range' for the operator.

There are four basic 'ranges' for operators as shown below.

1. Binary form. a b op where a and b are operands.
   op is one of the algebraic, relational or logical operators (except unary minus (-u) or .NOT.)
2. Unary form a op
   op is one of the operators -u, or .NOT.
3. 'Function, Array' form.
   a) Single argument (subscript) form
      F a op
   where F is the Function or Array name
   a is the argument
   op is the pseudo-operator f or a
   b) Multi-argument (subscript form)
      F a_1 a_2, a_3, - - -, op.
      For a two argument (subscript) form, this reduces to
      F a_1 a_2, op
4. 'Assignment' Form
   a) Simple assignment
      ae+ =
   b) multiple assignment
      a_1 a_2, a_3, - - - e =

where a, a_1, -- etc. are l.h.s. operands in the Assignment Statement. e is the r.h.s. operand in the Assignment Statement.

Knowing the operator, it is easy to determine the operands for any operation.

The Partial PN List - The Operation List

So far it has been shown how the complete PN equivalent can be generated for a FORTRAN expression.

In that the PN equivalent generates operators in their order of use, and in that all operands for an operator are already in the PN list when the operator is put into the list, it follows that for each operation all the information required for evaluating that operation is available at the time that the operator is transferred from the operator list to the PN list and that it is in fact not necessary to form the complete PN equivalent before starting to evaluate the FORTRAN expression.

In the FORTRAN Compiler, a partial PN list - the Operation List - is generated. This only contains operands and commas. When an operator would otherwise be passed to the PN list, it is used immediately to evaluate the operation. The result operand for the operation replaces all operands and commas used in the operation.

As an example of this Partial PN list generation, consider the evaluation of the expression A+B*(C-D). The complete PN evaluation has been given previously.

Step	Operation	Rule	List Contents	
			PNL	ΘL
1	A -> PNL	2	A	Empty
2	'+' -> ΘL	3.3	A	+
3	B -> PNL	2	AB	+
4	'*' -> ΘL	3.1	AB	+*
5	'(' -> ΘL	4	AB	+*(
6	C -> PNL	2	ABC	+*(
7	'-' -> ΘL	3.2	ABC	+*(-
8	D -> PNL	2	ABCD	+*(-
9	')' compared with ΘL			
	'-' -> PNL	5.1		
	(evaluate CD-)		ABR_1	+*(
	'(' removed from ΘL		ABR_1	+*
10	'E of S' compared with ΘL			
	'*' -> PNL	7		
	(evaluate BR_1*)		AR_2	+
	'+' -> PNL			
	(evaluate AR_2+)		R_3	Empty

R_1, R_2 and R_3 are the 'Result' operands for the evaluation of the operations CD -, BR_1* and AR_2+

(i.e. C - D, B*(C - D), and A + B * (C - D)).

The Actual Evaluation of Operations in FORTRAN

In the previous example, it was indicated that the evaluation of the expression required three operations, namely C-D, B*R_1, and A+R_2 where R_1 was the result of the operation C-D, and R_2 was the result of the operation A+R_2. The result of the overall expression was R_3.

The meaning of R_1, R_2, and R_3 depend on the way arithmetic operations are compiled by the FORTRAN compiler.

For the purpose of compilation, a 1900 series computer is considered to be a single accumulator (ACC), one index register (X3) machine.

ACC when defined as an operand or as the result of an operation contains a 'value'.

X3 when defined as an operand or as the result of an operation contains the 'address' of some storage area which contains a 'value'.

The basic operations defined so far have the following characteristics with regard to this assumption.

1. Binary Operations a b op

   Either 'a' or 'b' is ACC. The other operand is in store or is X3.

   The Result Operand is ACC.

2. Unary Operations a op

   'a' is ACC
   The Result Operand is ACC

3a. Function, Operation Fa_1, a_2,---, f

   F is Function Name
   a_1, a_2 --- etc. may not be ACC or X3
   The result operand is ACC

3b. Array Operation As_1s_2,--, a
A is the Array Name
s_1, s_2, --- etc. may not be ACC or X3
The result operand is X3

4. Assignment Operation a_1a_2,--, e=
a_1,a_2 may not be ACC
e is ACC
The result operand is ACC
although the actual result is a store location. (This result operand is rarely used)

Store Auxiliary Operations

Since an expression involves a series of operations, the result of one being an operand for the next, it can be seen that there is no guarantee that at the time it is required to perform an operation, the appropriate operands will be in the required form. In particular, unless the first operation is a Function Operation, at least one of the operands will be of the wrong form for the operation.

It is therefore necessary to define certain auxiliary operations which if the operands are of the wrong form may have to take place before the main operation.

The auxiliary operations to be defined are lac, lx3, sac, sx3. These have the following characteristics.

lac - Load ACC

Single Operand not ACC
Result Operand is ACC

lx3 Load X3

Single Operand is Tx or 'Store' Operand
Result Operand is X3

sac Store ACC

Single Operand is ACC
Result Operand is Ta

sx3 Store X3

Single Operand is X3
Result Operand is Tx

Ta is a Temporary Storage Area used to store ACC if it is not required for the current operation.

Tx is a Temporary Storage Area used to store X3 if it is not required for the current operation.

This set of operations are the 'Store Auxiliary' operations of the Compiler.

The operations lx3 and sx3 do not destroy ACC

The operations lac and sac do not destroy X3

Consider again the example A+B*(C-D)

This had previously been broken down into the operations as shown below.

	Operation List	Operation Performed	Total Operation
1.	ABCD-	$C-D \rightarrow R_1$	C-D
2.	ABR_1*	$B*R_1 \rightarrow R_2$	B*(C-D)
3.	AR_2+	$A+R_2 \rightarrow R_3$	A+B*(C-D)

The FORTRAN Compiler treats this in more detail according to the rules governing operands for the various operators as follows

Operation List		Operation Performed	Total Operation
1.	ABCD(lac)-	D → ACC	D
2.	ABCACC -	C - ACC → ACC	C-D
3.	ABACC*	B*ACC → ACC	B*(C-D)
4.	AACC+	A+ACC → ACC	A+B*(C-D)

The operator lac was used to put the operands into a standard form for the binary operation '-'. It operates on the top item in list, namely D.

Consider the following example

A*B + C*D

The FORTRAN Compiler breaks this down to the following set of operations.

Operation List		Operation Performed	Total Operations
1.	AB(lac)*	B → ACC	B
2.	AACC*	A*ACC → ACC	A*B
3.	ACC CD (sac)(lac)*	ACC → Ta_1	A*B → Ta_1
4.	Ta_1 CD (lac)*	D → ACC	D
5.	Ta_1 C ACC*	C*ACC → ACC	C*D
6.	Ta_1 ACC +	Ta_1 + ACC → ACC	A*B + C*D

At steps 1 and 4, the operator lac is used to produce an operand in the standard form for the binary operation*.

At step 3 it is necessary to use the operator sac to store ACC in a temporary storage location Ta_1 becuase, although ACC is in use it is not one of the operands for the current operation.

The operator sac does not work on the top item, but on the item which is ACC. The item is replaced by the result operand Ta (Ta_1 in this case).

The fact that sac works only on the item which is ACC, and that there may not be two ACC items in the list at the same time requires (and enables) the compiler to keep a pointer to the current list item for ACC. If no item is ACC, then this pointer is clear.

Consider the following example

A(I) + B*C where A is an array name.

The FORTRAN Compiler will break this down to the following set of operations.

Operation List	Operation Performed	Total Operation
1. AIa	A(I) → X3	A(I) in X3
2. X3 BC (lac)*	C → ACC	C
3. X3 B ACC*	B*ACC → ACC	B*C
4. X3 ACC +	A(I) + ACC → ACC	A(I) + B*C

Here the operation B*C is defined as not requiring or destroying X3 so there is no requirement for a SX3 operation at step 2.

On the other hand in the expression

A(I) + F(X) where A is an array name and F is a Function.

The following operations are compiled.

1.	AIa	A(I) → X3	A(I) in X3
2.	X3 FX sx3 f	X3 → Tx_1	X3 → Tx_1
3.	Tx_1 FX f	F(x) → ACC	F(x)
4.	Tx_1 ACC +	Tx_1 + ACC → ACC	A(I) + F(X)

In this example, the Function operation is defined as possibly destroying X3 which, since it holds an operation result, must be stored using the operator sx3. Note that as for the operator sac the operator sx3 does not work on the top item but on the item which is X3. Therefore as for sac, the compiler is required to keep a pointer to the current list item for X3. If no item is X3 then this pointer is clear.

Operation Modes - Mode Auxiliary Operations

In FORTRAN, an operand of an expression will be defined as having one of five 'modes' (INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL)

As the actual representation in the compiled program of an operand depends on the operand mode, and the instructions compiled for an operation depend on the mode of the operands, it is necessary to give the operand mode information in the Operation List. The mode of the result of an operation is defined by the modes of the original operands and by the operator. The result mode is not necessarily the same as that of the operands.

The modes of the operands have to be in a standard form for an operation to be defined.

The standard operand modes are given in the following table.

1 a. Binary Operators +,-,*,/ abop

'a','b' must be of the same mode (not LOGICAL)
The Result is of the same mode as the operands

b. Binary Operator ** ab**

'b' must be INTEGER if 'a' is INTEGER or COMPLEX
'b' must be REAL or INTEGER if 'a' is REAL or DOUBLE PRECISION
Neither 'a' nor 'b' may be LOGICAL
The Result is the mode of 'a'

c. Binary Operators .EQ.,.NE.,.LE.,.GE.,.LT.,.GT. abop

a,b must be of the same mode (not Logical)
The Result is LOGICAL mode

d. Binary Operators .AND.,.OR. ab op

a,b must be LOGICAL mode
The Result is LOGICAL mode

2. a. Unary Operator '$-_u$' $a-_u$

'a' must be any mode except LOGICAL
The Result is the same mode as 'a'

b. Unary Operator .NOT. a.NOT.

'a' must be LOGICAL mode
The Result is LOGICAL mode.

3. a. Function Operator f Fa_1---f

a_1,--- may be any mode
The Result is the mode of F

b. Array Operator a As_1---a

s_1,--- must be INTEGER mode
The Result is the mode of A.

4. Assignment Operator = $a_1,a_2,---,e=$

The modes of a_1,--- and e are the same.
The Result is the mode of a_1,

If the modes for the operands in an operation are not standard certain auxiliary operations may have to be performed to make the operand modes standard before the main operation can be performed. Certain combinations of modes may produce an undefined operation. This will cause the Compiler to indicate an Error.

Consider the following examples.

The subscripts R, I, and D indicate that the operand is REAL, INTEGER, or DOUBLE PRECISION.

Example 1. $A_R + B_R$

In this operation the modes are both the same and standard for the operator '+'. Therefore the operation is "immediately definable" (after doing the auxiliary lac operation on B_R)

1. $B_R \rightarrow ACC_R$ (lac)

2. $A_R + ACC_R \rightarrow ACC_R$ +

Example 2. $A_R + B_I$

In this operation the modes are not the same and therefore are not standard for the operator '+'. The operation (after having done the lac operation) is not 'immediately definable'. An auxiliary operation 'Float ACC' (fa) must be performed on B before the operation becomes standard.

The fa operation takes an INTEGER operand in ACC and returns a REAL result in ACC

1. $B_I \rightarrow ACC_I$ (lac)

2. $ACC_I \rightarrow ACC_R$ (fa)

3. $A_R + ACC_R$ +

Example 3. $A_D + B_I$

Once again an auxiliary operation is required to bring the operands to a standard form. In this case, two auxiliary operations are required, fa and da.

The da operator takes a REAL operand in ACC and returns a DOUBLE PRECISION result in ACC.

1. $B_I \longrightarrow ACC_I$ (lac)

2. $ACC_I \longrightarrow ACC_R$ (fa)

3. $ACC_R \longrightarrow ACC_D$ (da)

4. $A_D + ACC_D$ +

The two auxiliary operations were required to put 'B' in Double precision form. In theory, only one operation is required (INTEGER $\longrightarrow$ DOUBLE PRECISION) but such an operation is not in the set of operations currently available as auxiliaries in the compiler.

The mode changing auxiliary operations used by the compiler are outlined below. They all use one operand and give one result.

Operator	Operand	Result	Remarks
fa	ACC_I	ACC_R	Float INTEGER ACC
fs	$X3_I$	$X3_R$	Float Integer Quantity not in ACC
da	ACC_R	ACC_D	Convert REAL ACC to Double Precision.
ds	$X3_R$	$X3_D$	Convert REAL Quantity, not in ACC, to Double Precision.
ca	ACC_R	ACC_C	Convert REAL ACC to Complex
cs	$X3_R$	$X3_C$	Convert REAL Quantity, not in ACC, to Complex
ifa	ACC_R	ACC_I	Convert REAL ACC to Integer
ida	ACC_D	ACC_R	Convert DOUBLE PRECISION ACC to Real.

The operations using fa, da, ca, ifa and ida do not destroy X3.
The operations fs, ds and cs do not destroy ACC.

Temporary Store Allocation

Consider the compilation of the following Assignment statement.

X = A*B+SQRT(X+Y)*(X-Y)

It is assumed that the mode of all items is the same, and is REAL.

The 'Compilation Table' which follows is divided into the following Columns.

1. Step. This gives the number of main and auxiliary operations which have been done (including the current operation).

2. Markers. These give the values of the X3 and ACC marker. They contain the addresses in the PN List in which the X3 operand and the ACC operand appear. If X3 or ACC do not appear in the PN List, then the value of the associated marker will be zero.

3. Op This gives the main operator which is being processed.

4. PN List. This contains the items which are in the Operand List at the time of the start of a main or auxiliary operation.
The sequence of numbers along the top give the "address" of the items in the List, and are, in the Compilation Table, associated with the values in the Markers.

5. Aux Op. This gives the auxiliary operation which is currently being performed. '-' in the field indicates that the main operation is being done.

Original Statement.

X = A*B+SQRT(X+Y)*(X-Y)

Step	Markers X3	Markers ACC	Op	PN List 1	2	3	4	5	6	Aux Op	Comments
1	0	0	*	X	A	B				lac	B → ACC
2	0	3	*	X	A	ACC				-	A*B → ACC
3	0	2	+	X	ACC	SQRT	X	Y		sac	ACC → Temporary Storage 1
4	0	0	+	X	Ta_1	SQRT	X	Y		lac	Y → ACC
5	0	5	+	X	Ta_1	SQRT	X	ACC		-	X+Y → ACC
6	0	4	f	X	Ta_1	SQRT	ACC			sac	ACC → Temporary Storage 2
7	0	0	f	X	Ta_1	SQRT	Ta_2			-	SQRT(X+Y) → ACC
8	0	3	-	X	Ta_1	ACC	X	Y		sac	ACC → Temporary Storage 3
9	0	0	-	X	Ta_1	Ta_3	X	Y		lac	Y → ACC
10	0	5	-	X	Ta_1	Ta_3	X	ACC		-	X-Y → ACC
11	0	4	*	X	Ta_1	Ta_3	ACC			-	SQRT(X+Y)*(X-Y) → ACC
12	0	3	+	X	Ta_1	ACC				-	A*B+SQRT(X+Y)*(X-Y) → ACC
13	0	2	=	X	ACC					-	X=A*B+SQRT(X+Y)*(X-Y)

The original statement breaks down into 13 separate immediately definable operations. Certain of these operations involve the use of Temporary Storage Areas to store intermediate results (Steps 3, 6 and 8). In the table these areas have been called Ta_1, Ta_2 and Ta_3.

The size of a storage area depends on its use. The 'Ta' areas are 2 or 4 words long depending on the mode of the operand for the sac operation. The 'Tx' areas are 1 word long regardless of the mode of the operand for the SX3 operation.

In compiled program these areas occupy a single area of store (LW area) and therefore a working store assignment rule is required by the compiler. A marker (CWSA) is required to determine the address of the next Temporary area which may be assigned.

A simple Temporary store assignment rule would be following.

1. When a Temporary store is required, the address of the area is given by the current value of CWSA. The value of CWSA is then incremented by the size of the area required.

By this rule, the following Temporary store assignment would be given in the previous example. Assuming that the original value of CWSA was To, and that each 'Ta' requires 2 locations,

Then $Ta_1 = To$
$Ta_2 = To + 2$
$Ta_3 = To + 4$

Thus 6 Temporary store locations are required for the statement.

This is obviously extravagant in storage space, and in a complicated assignment statement could lead to considerable waste space.

Due to the rules for arithmetic used in the current FORTRAN Compiler, a Temporary store location is only used in one operation and can therefore be reused once that operation has been processed.

In the previous example, the Temporary area Ta_2 was made at Step 6 and was used in the operation of Step 7. It then follows that the area Ta_3 generated at Step 8 can be the same as area Ta_2.

The rules for Temporary Store Assignment actually used in the compiler as follows.

1. When a temporary store is generated by an auxiliary operation, the address of the area is given as the current value of CWSA. The value of CWSA is then incremented by the size of the area.

2. When a temporary store is used in an operation, CWSA is given the address of that temporary store.

Following again the previous example, the value of CWSA after the various steps is given below.

Start.	CWSA = To			
Step 3	CWSA = To + 2	Ta_1 = To		
Step 6	CWSA = To + 4	Ta_1 = To	Ta_2 = To + 2	
Step 7	CWSA = To + 2	Ta_1 = To		
Step 8	CWSA = To + 4	Ta_1 = To	Ta_3 = To + 2	
Step 11	CWSA = To + 2	Ta_1 = To		
Step 12	CWSA = To			

At the end of evaluation of an expression the value of CWSA should be the same as it was at the start. Only in certain peculiar circumstances (generation of Statement Functions) is this not the case.

The rules stated above for Temporary Store assignment will only work if one can guarantee to use the Storage areas in the inverse order to which they were assigned. It can be seen by an example that with the auxiliary operations as described to this point that this is not necessarily the case.

In the following example, a Temporary Store item is designated by 'T' followed by 'a' or 'x' followed by an integer.

'a' indicates that the Temporary Store is holding a value and was generated by a sac operation
'x' indicates that the Temporary Store is holding an 'address' and was generated by a SX3 operation.

The Integer indicates the order of assigning Temporary Storage areas.

Example

Original Expression (Assignment Statement)

$$Y = A(I) + (B + C) * ((D + E) * (A(J) + F) + ((G + H) + FN(X)))$$

The main operations which are used have the following characteristics with regard to ACC and X3

Array Op. (a)	destroys ACC and X3	leaves result as X3
+,*,=	destroys ACC	leaves result as ACC
Function op (f)	destroys ACC and X3	leaves result as ACC

In this example 'A' is an array name. 'FN' is a function name

Step	Markers		op	PN List								Aux	Comments
	X3	ACC		1	2	3	4	5	6	7	8	Op	
1	0	0	a	Y	A	I						-	A(I)→ X3
2	2	0	+	Y	X3	B	C					lac	C → ACC
3	2	4	+	Y	X3	B	ACC					-	B+C → ACC
4	2	3	+	Y	X3	ACC	D	E				sac	ACC → Ta_1
5	2	0	+	Y	X3	Ta_1	D	E				lac	E → ACC
6	2	5	+	Y	X3	Ta_1	D	ACC				-	D+E → ACC
7	2	4	a	Y	X3	Ta_1	ACC	A	J			sac	ACC → Ta_2
8	2	0	a	Y	X3	Ta_1	Ta_2	A	J			sx3	X3 → Tx3
9	0	0	a	Y	Tx3	Ta_1	Ta_2	A	J			-	A(J) → X3
10	5	0	+	Y	Tx3	Ta_1	Ta_2	X3	F			lac	F → ACC
11	5	6	+	Y	Tx3	Ta_1	Ta_2	X3	ACC			-	A(J) + F → ACC
12	0	5	*	Y	Tx3	Ta_1	Ta_2	ACC				-	(D+E) * ACC → ACC
13	0	4	+	Y	Tx3	Ta_1	ACC	G	H			sac	ACC → Ta4
14	0	0	+	Y	Tx3	Ta_1	Ta_4	G	H			lac	H → ACC
15	0	6	+	Y	Tx3	Ta_1	Ta_4	G	ACC			-	G + H → ACC
16	0	5	f	Y	Tx3	Ta_1	Ta_4	ACC	FN	X		sac	ACC →Ta_5
17	0	0	f	Y	Tx3	Ta_1	Ta_4	Ta_5	FN	X		-	FN(X) → ACC
18	0	6	+	Y	Tx3	Ta_1	Ta_4	Ta_5	ACC			-	(G+H) + FN(X)→ ACC

Assuming the initial value of CWSA was To, and that Tx areas occupy 1 word and that Ta areas occupy two words, the following Temporary area allocations would be made at the end of the steps shown.

Start	CWSA = To			
Step 4	CWSA = To + 2	Ta_1 = To		
Step 7	CWSA = To + 4		Ta_2 = To + 2	
Step 8	CWSA = To + 5			Tx_3 = To + 4
Step 12	CWSA = To + 2		Ta_2 used	
Step 13	CWSA = To + 4		Ta_4 = To + 2	
Step 16	CWSA = To + 6			Ta_5 = To + 4

It can be seen that the sac operation of step 16 assigned the same starting address to Temporary area Ta_5 as to the Temporary area Tx3. Thus the information stored at step 8 is lost.

The problem arises because the working store areas are not used in the inverse order to which they were assigned (eg. Ta_2 was used before Tx3).

The problem is solved in the FORTRAN compiler by making the sac operator and the sx3 operator slightly more complex as follows.

1. sac — The operand is the PN list item pointed to by the ACC marker. If the X3 marker is set to a smaller PN list address than the ACC marker at the time a sac operation is to be performed, a sx3 operation is performed first.

2. sx3 — The operand is the PN list item pointed to by the X3 marker. If the ACC marker is set to a smaller PN list address than the X3 marker, at the time a sx3 operation is to be performed, a sac operation is performed first.

In the previous example, this is shown as follows.

Step	Markers		Op								Aux Op		Comments
	X3	ACC		1	2	3	4	5	6		1	2	
1	0	0	a	Y	A	I					-		A(I) → X3
2	2	0	+	Y	X3	B	C				lac		C → ACC
3	2	4	+	Y	X3	B	ACC				-		B + C → ACC
4	2	3	+	Y	X3	ACC	D	E			sac	sx3	X3 → Tx_1
5	0	3	+	Y	Tx_1	ACC	D	E			sac	-	ACC → Ta_2
6	0	0	+	Y	Tx_1	Ta_2	D	E			lac		E → ACC
7	0	5	+	Y	Tx_1	Ta_2	D	ACC					D + E

At step 4, the value of the X3 marker was less than the value of the ACC marker and so the simple sac operation was preceded by a sx3 operation. In this way the Temporary areas are assigned in the inverse order to their use in subsequent operations.

Operands — At source language level, the FORTRAN Compiler is dealing with 'names' and 'constants' as operands. During the evaluation of an expression it is also processing Temporary Storage operands (Tx and Ta) as well as Acc and X3.

Excluding Function names, an operand will generally be associated with a storage area in the final compiled program and is manipulated within the compiler, not by name, but by the address of that area. Most variable names and constants are of this type as are Ta operands. These are called 'Store' operands.

Some operands are associated indirectly with a storage area by means of words which themselves contain the address of the relevant storage areas. The X3 operand and the Tx operand are of this type. These are called 'Indirect' operands.

If the operand is a small integer (<4096), for some classes of operation the compiler manipulates the operand by its value rather than by the address of a word containing the constant. This type of operand is a 'Value' operand.

Relationship between Operators and Operands

By prior use of Store Auxiliary operations, the operands for any main operator can be put into a standard form (e.g. for Binary operations one operand is Acc and one is either a 'Store' operand, X3 or a 'Value' operand)

By investigating the mode, position and type of each operand along with the relevant operator, an operation can be completely defined. Wherever possible in the FORTRAN Compiler, the 'characteristics' of an operation are associated with the operator as shown below.

1. Arithmetic Relational and Logical Operations

1.1 Arithmetic Binary Operations (+, -, *, /, **)

These operations can always be put into the form a Acc op or Acc b op. These forms define two distinct classes of operation - the 'reverse' (a Acc op) and the 'forward' (Acc b op) classes. For the operators '+' and '*', these classes are identical, a fact which is not used in the compiler. These two classes will be denoted by the letters 'r' and 'f'. e.g. '+(f)' implies an operation of the form Acc b +

The operands may have the same or different modes. Each combination of mode defines a separate subclass of operation within the main classes just described. If the modes are indicated by the letters 'I' - INTEGER, 'R' - REAL, 'D' - DOUBLE PRECISION, 'C' - COMPLEX, and 'L' - LOGICAL, these subclasses can be described by associating two letters (representing the mode of Acc and the mode of the other operand) with the operator. Thus an operator presented as '+ (f,I,R)' would imply an operation of the form Acc b + where 'Acc' is of mode INTEGER and 'b' was of mode REAL.

The 'type' of the non-Acc operand defines another subclass of operation depending on whether the type is 'Store'(S), X3 (X), or 'Value'(V).

An operator represented as '+(f,I,R,S)' would completely define the operation to be of the form Acc b + where Acc is of mode INTEGER and 'b' is a 'Store' operand of mode REAL.

In general an algebraic binary operation can be defined by a construct of the form

$$a\ b\ op\ (d,\ m_A,\ m_o,\ t)$$

where a, b are the operands
d - direction of Operation
m_A - mode of Acc
m_o - mode of other operand
t - Type of other operand

1.2 Relational Binary Operations (.LE.,.GE.,.EQ.,.NE.,.GT.,.LT.)

These operations of the form a b op are considered by the compiler to be two distinct operations of the form ab- (or ba-) followed by a unary operation on the result.

The result of the '-' operation is Acc. The unary operation following this operation is a function only of the original relational operator and is independent of the mode of Acc.

Thus a complete Relational Binary operation is defined by a construct of the form

$$a\ b\ -\ (d,\ m_A,\ m_o,\ t)\ op'$$

where a, b are the operands
d, m_A, m_o, t are as defined for Algebraic Binary operations
op' is a unary operation which depends only on the original Relational Operator.

1.3 Logical Binary (.AND.,.OR.)

These operations can always be put in the form a Acc op, or Acc b op. The modes of Acc and the other operand are the same and must be LOGICAL. The non-Acc operand may be either a 'Store' operand or X3.

Thus a logical Binary operation can be defined by a construct of the form

a b op (d,t)

where a, b are the operands
d - direction of Operation
t - Type of non Acc operand (S or X)

1.4 Unary Minus ($-_u$)

This operation is of the form Acc-$_u$ where Acc may be of any mode other than LOGICAL

Thus a unary minus operation can be defined by a construct of the form

a -$_u$(m_A)

where a is Acc
m_A - mode of Acc

1.5 Logical .NOT.

This operation is of the form Acc op where Acc may be of mode LOGICALonly Thus there is only one .NOT.operation.

2. Store Auxiliary Operations

These operations involve only one operand and so are conceptually of the form a op although the operand is not necessarily the top item in the Operation List.

2.1 Store Accumulator Operations (sac)

These operations put Acc into a storage area. They are a function of the 'mode' of Acc only.

Thus all operations using this operator can be described by a construct of the form

Acc sac (m_A)

where m_A = mode of Acc

2.2 Store X3 Operation (sx3)

This operation is independent of the mode of X3 and involves storing X3 in a Tx operand.

There is only one sx3 operation.

2.3 Load Acc Operations (lac)

These operations take a 'Store', X3, Tx or 'Value' operand and produce a 'Value' in Acc in the standard form required for main operators.

The actual operation depends on the mode and type of the operand.

Thus all operations using this operator can be described by a construct of the form

a lac (m_o, t)

where m_o = mode of operand
t = type of operand

2.4 Load X3 Operations (lx3)

There are two operations in this class which depend on whether the operand is a 'Store' or a Tx operand. There are no 'Value' operands for this class.

Thus the operation is described by a construct of the form

a lx3 (t)

where t = type of operand (S or Tx)

3. Function and Array Operations

The standard form for Function or Array operations allows the arguments or subscripts to be 'Store' or Tx type operands. Other types as yet not mentioned (Function and Array names) are also allowed but these will not be discussed at this point.

The operations are independent of the mode of the Function or Array name, but depend on the type of operand of each argument or subscript.

Once the operands are in a standard form for a Function or Array operation, the operation is treated as a composite series of simple operations which generate first the CALL to the Function or to the Array routine and then the argument or subscript list.

Summary of Operator Associations

The following table summarizes the operator associations of the previous section.

Operation	Operators	Operation Form	Operator Association
Arithmetic Binary	+,-,*,/,**	a b op	op (d,m_A,m_o,t)
Relational Binary	.LE.,.LT.,.EQ. .NE.,.GE.,.GT.	a b op	-(d,m_A,m_o,t) then op'
Logical Binary	.AND.,.OR.	a b op	
Unary Minus	$-_u$	a $-_u$	$-_u$ (m_A)
Logical .NOT.	.NOT.	a.NOT.	.NOT.
Store Acc	sac	Acc sac	sac (m_A)
Store X3	sx3	X3 sx3	sx3
Load Acc	lac	a lac	lac (m_o,t)
Load X3	lx3	a lx3	lx3 (t)

d - direction of binary operation f = Acc b op
r = a Acc op

m_A,m_o - mode of Acc, mode of operand (not Acc)

t - type of Operand s - store
x - x3) Indirect
Tx -)
V - Value

Practical Compilation

So far, this discussion has been in general terms without any particular reference to the actual way in which these operators and their associations are interpreted within the compiler, so as to generate actual instructions. The following sections of this document will show in some detail how the various operations are actually compiled.

Operation Compilation (I)

When an operator is transferred from the Operator List, it is classified as to type - Arithmetic Binary, Relational Binary, Logical Binary, Unary minus, Logical.NOT., Function and Array reference, Assignment operations etc. Each type of operation is processed by separate routines, although these routines use common subroutines to generate instructions and perform auxiliary operations if necessary. The operation types that will be described here are the following.

1. Binary Operations (Arithmetic and Logical)
2. Relational Binary
3. Unary Minus
4. Logical.NOT.
5. Assignment (=)
6. Store Auxiliary

1. Binary Operations (Arithmetic and Logical)

A Binary operation uses as operands the top two items in the PN list. There is no guarantee when starting to evaluate a binary operation that these top items are in the standard form required to satisfactorily compile the operation. To ensure standard conditions, a routine BINSTANDARD is used. This performs all required Store auxiliary operations.

1.1 Arithmetic Binary Operations

As indicated in the 'Summary of Operator Associations' an arithmetic binary operation can be defined by the following parameters

1 op - operator
2 d - direction of Operation Acc b op, a Acc op
3 m_A - mode of Acc
4 m_o - mode of 'a' or 'b'
5 t - type of operand

Not all values of M_A, M_o are allowed for all operators or operand type. In particular neither m_A or m_o may be of mode LOGICAL. Also, if 't' is a 'Value' operand (i.e. a small integer), then m_o may only be of type INTEGER and is therefore redundant.

In order to compile instructions, these parameters are used to address a table entry, the entry indicating whether the particular operation described is valid and if so, what instructions are to be compiled for the operation. The details of this phase of compilation will be described in a later section.

If no cognisance was taken of the redundancies and of the large numbers of invalid mode-operator-etc combinations, a table with 480 entries would be required to describe all possible Arithmetic Binary operations. By separating out operations as indicated below, the number of table entries required is reduced to 250.

The Parameter Groups separated out for Arithmetic Binary Operations are the following

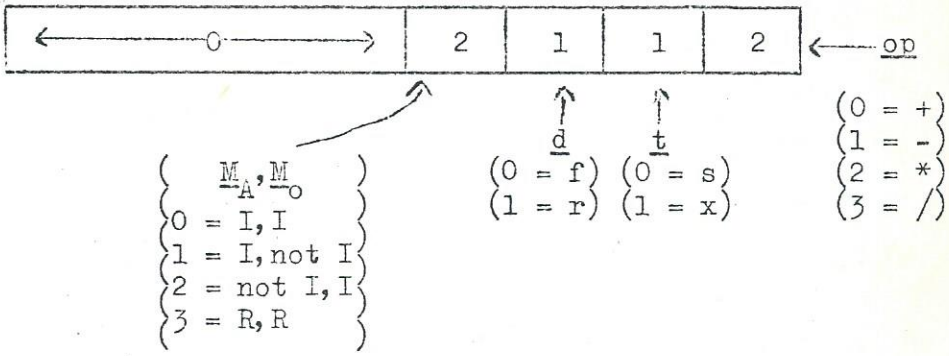
1.1.1 Standard Binary

op = +, -, *, /

t = s,x

modes M_A = I, M_o = I,R,D,C
or M_A = R, M_o = I,R
or M_A = D,C, M_o = I

The position in the group is defined by a binary value as shown in the following diagram.



Thus an operation Acc b * with M_A = R and M_o = I has a value of 18 as defined in this group.

Note that if one (but not both) operand mode is I, then there are 3 operations which are defined to be identical, namely those in which the other operand is REAL, DOUBLE PRECISION or COMPLEX. This is due to the lack of any Mode Auxiliary operations for integer items except fa and fs (see 'Mode Auxiliary Operations' -p.16)

1.1.2 DOUBLE PRECISION Binary

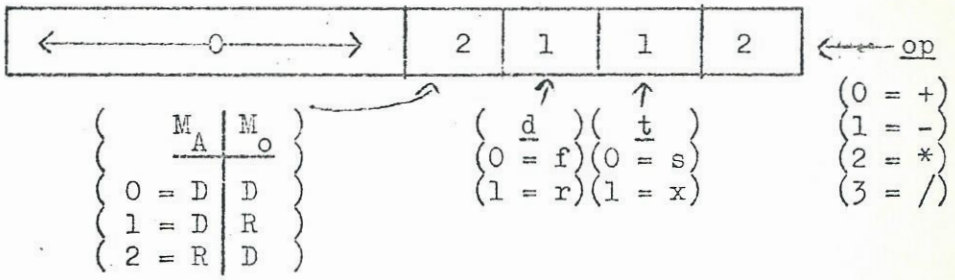
op = +, -, *, /

t = s, x

modes $M_A = D, M_o = R,D$

or $M_A = R, M_o = D$

The position in the group is defined by a binary value as shown in the following diagram



This is identical in form to that for Standard Binary. The only difference is in the interpretation of the two mode bits.

As an example, the operation a Acc + with M_A, M_o both 'D', has a value of 8 as defined by this group.

1.1.3 COMPLEX Binary

op = +, -, *, /

t = s, x

modes $M_A = C, M_o = R,C$

or $M_A = R, M_o = C$

The position in this group is defined in an identical manner to that for DOUBLE PRECISION Binary except that the interpretation of the two mode bits is as follows.

0 - $M_A = C, M_o = C$

1 - $M_A = C, M_o = R$

2 - $M_A = R, M_o = C$

1.1.4 Small Integer Binary

op = +, -, *, /

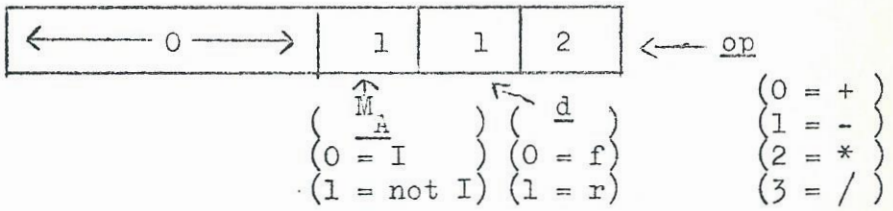
t = V

modes M_A = I,R,D,C

M_o = I

For this group of operations, the non-Acc operand is a small integer ('Value' operand). The mode of this operand is INTEGER by definition and therefore need not be explicitly mentioned.

The position in the group is defined by a binary value as shown in the following diagram



Thus the operation Acc 3 + with M_A = I is given a value 0 for this group.

1.1.5 Standard Exponentiation

op = **

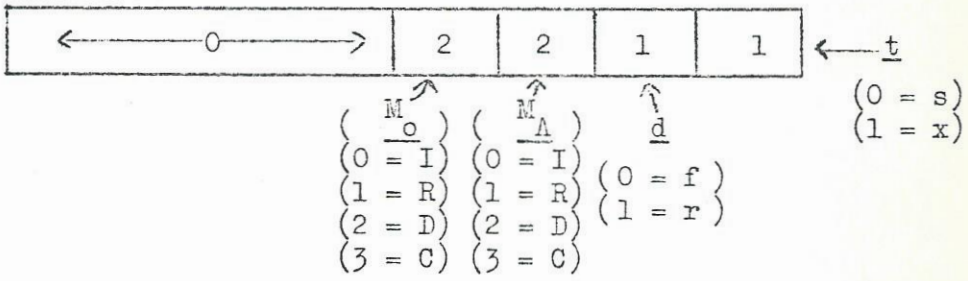
t = s,x

modes M_A = I,R,D,C

M_o = I,R,D,C

This group of operations contains all the exponentiation operations except for those with a 'Value' operand.

The position in the group is defined by a binary value as shown in the following diagram



Thus the operation a Acc ** for a-REAL and Acc-INTEGER is given the value 16 for this group.

1.1.6 Small Integer Exponentiation op = **

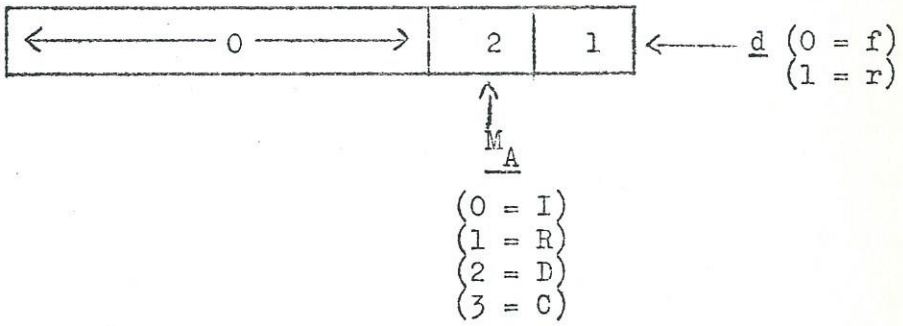
t = V

modes M_A = I,R,D,C,

M_o = I

For this group of operations, the non-Acc operand is a small integer ('Value' operand). The mode of this operand is INTEGER by definition and therefore need not be explicitly mentioned.

The position in the group is defined by a binary value as shown in the following diagram



Thus the operation Acc N op with M_A = R has the value 2 for this group.

1.2 Logical Binary Operations

As indicated in the 'Summary of Operator Associations' a logical binary operation can be defined by the following parameters

1. op - operator
2. d - direction of Operation Acc b op, a Acc op
3. t - type of operand

The logical binary operations are symmetric. Thus as far as compiled instructions are concerned, the direction of operation is immaterial.

e.g. A.AND.Acc is the same operation as Acc.AND.A.

At present, there is no 'Value' operand which can be used with logical operators. Thus there are only two types of operands, s, and x.

There are only two operators which produce logical binary operations - . .AND. and .OR.

There are therefore only four logical binary operations, which are grouped together and given values as follows

0 op = .AND.; t = s
1 op = .AND.; t = x
2 op = .OR. ; t = s
3 op = .OR. ; t = x

Thus the operation X3 Acc.OR. would have the value 3 for this group.

2. Relational Binary Operations

A Relational Binary operation uses as operands the top two items in the PN list. Depending on the actual operator, the routine processing Relational Binary operations may interchange the top two items. If one of the two items is Acc, the Acc pointer is changed to point to the new location of Acc.

At this stage the routine which processes arithmetic binary operations (BINARITH) is entered to compile an operation using the two top PN items as operands with '-' as the operator. On exit from this routine, the top PN item will be Acc (the result of an operation of the form 'ab-').

The Relational Binary operation is completed by performing a unary operation on Acc. This operation leaves a LOGICAL result as Acc. The operation is independent of the mode of Acc but depends on the original relational operator.

These operations can be defined as a group, the position in the group being as shown in the following list.

0 = .LT.
1 = .LE.
2 = .EQ.
3 = .NE.
4 = .GT.
5 = .GE.

3. Unary Minus

This operation requires one operand (the top item in the PN list) which must be Acc. At the time a unary minus operation is to be compiled, since there is no guarantee that the operand will be Acc, a routine UNSTANDARD is entered to compile all required Store Auxiliary operations.

As indicated in the 'Summary of Operator Associations', a unary minus operation can be defined by one parameter - the mode of Acc (M_A).

Thus the unary minus operations can be defined as a group, the positions in this group being defined as follows

0 - M_A = INTEGER
1 - M_A = REAL
2 - M_A = DOUBLE PRECISION
3 - M_A = COMPLEX
4 - M_A = LOGICAL

Although there is a position in the group for Acc in LOGICAL mode, it represents an illegal operation, this is found as such by the operation instruction generating routine.

4. LOGICAL .NOT.

This operation requires one operand (the top item in the PN list) which must be Acc at the time the actual .NOT. operation is to be compiled. As there is no guarantee that the operand will be Acc a routine UNSTANDARD is entered to compile all required Store auxiliary operations.

Acc may only be of LOGICAL mode for this operation. Thus there is only one .NOT. operation.

5. Assignment Operations

a. Simple Assignment (a = e)

For simple assignment, the Assignment operation uses the top two items in the PN list as operands. It is necessary in an Assignment operation for the top item in the PN list to be Acc. To ensure this prior to the actual generation of the Assignment operation, store auxiliary operations are performed if required.

At this stage, the non-Acc operand is either of type 'S' or 'X'.

The Assignment operations require the following three parameters

1. $\underline{t}$
2. M_A
3. M_o

$\underline{t}$ = s,x

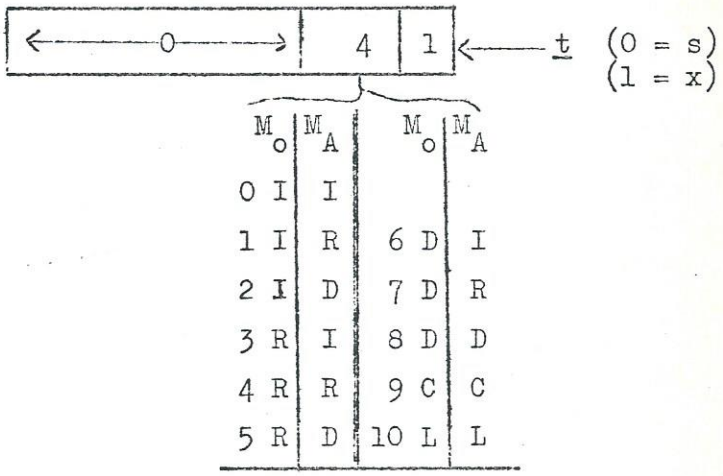
for M_A = I,R,D ; M_o = I,R,D,

for M_A = C; M_o = C

for M_A = L; M_o = L

All allowable combinations produce a group of 22 Assignment operations.

The position of an operation in this group is defined by a binary value as shown in the following diagram



M_o	M_A	M_o	M_A
0 I	I		
1 I	R	6 D	I
2 I	D	7 D	R
3 R	I	8 D	D
4 R	R	9 C	C
5 R	D	10 L	L

In the PN list, the result of a simple Assignment operation is Acc with the same mode as the original non-Acc operand

i.e. $M_A' = M_o$.

The position of Acc in the PN list is the same as the original position of the non-Acc operand.

Thus the operation 'a Acc =' with M_A = I, M_o = R has a value of 6 in this group of operations. In the PN list the result of the operation is Acc (overwriting 'a') with M_A = R.

b. Multiple Assignment ($a_1, a_2, ---a_k$ = e)

The operation of multiple Assignment is a series of simple Assignment operations.

In the PN list, for multiple Assignment, the top two items will be ', Acc' rather than 'a Acc'. Physically moving Acc down one item (overwriting the ',') and amending all associated pointers (including the PN list pointer) will then produce a simple operation of the form 'a Acc ='. This simple operation is then evaluated, and the resultant top two items of the PN list are checked to see whether the top two items are in the form for multiple or simple Assignment.

If the form is still that for multiple Assignment the process is repeated. If the form is that for simple Assignment, the whole operation is terminated as soon as the simple Assignment operation is completed.

Example If the Assignment produces the following PN list a b,c, Acc the operations proceed as follows

1.	Move Acc	a b, c Acc
2.	do C = Acc	a b, Acc
3.	Move Acc	a b Acc
4.	do b = Acc	a Acc
5.	do a = Acc	Acc

The resultant operations performed for a,b,c = Acc are c = Acc, b = c, a = b

6. Store Auxiliary Operations

There are basically four types of Store Auxiliary operations, lac, sac, lx3 and sx3 operations.

6.1 Load Acc (lac)

As indicated in the Operator Association table, a lac operation can be defined by two parameters

1. M_o
2. t

t may be one of four types - S,X,Tx,V

For each of these types (except V), M_o may be I,R,D,C or L giving a total of 16 operations (including V)

These define a group. The value of an operation within this group is given in the following table

	M_o	t		M_o	t
0	I	S	8	L	S
1	I	Tx	9	L	Tx
2	R	S	10	I	V
3	R	Tx	11	I	X
4	D	S	12	R	X
5	D	Tx	13	D	X
6	C	S	14	C	X
7	C	Tx	15	L	X

6.2 Store Acc (sac)

A sac operation can be defined by one parameter, M_A.

This defines a group of 5 operations. The value of an operation within this group is shown below

0 - M_A = I
1 - M_A = R
2 - M_A = D
3 - M_A = C
4 - M_A = L

6.3 Load X3 (lx3)

A lx3 operation depends only on the type of the operand. This may be either S or Tx, giving a total of two operations.

6.4 Store X3 (sx3)

There is only one sx3 operation. This produces a Tx result.

Group Identifiers

Within the compiler, at the time that an operation is defined, the operation is presented to the 'Instruction Generator'as a group number and a position number.

The groups are numbered as follows

1. Main Operations
   - 0 - Standard Binary
   - 2 - Double Precision Binary
   - 4 - Complex Binary
   - 6 - Small-Integer Binary
   - 8 - Standard Exponentiation
   - 10 - Small Integer Exponentiation
   - 16 - Unary Minus
   - 20 - Relational Binary
   - 22 - Logical Binary
   - 24 - Assignment.

2. Store Auxiliary Operations
   - 12 - Load ACC (lac)
   - 14 - Store ACC (sac)

The operations .NOT., lx3, and sx3 are not associated with groups in the compiler because there is only one .NOT., and one sx3 operation. There are only two lx3 operations. These operations are presented to the Instruction Generator in a different way.

Thus any of the operations defined in a group can be described by using two integers (g,p) to specify the group and position. These are used by the Instruction Generation routine (ACOMPIL, Entry ACOM) to extract the correct complete and skeleton instructions for the operation, complete the skeleton instructions and then output them along with the appropriate instruction relativisors.

A skeleton instruction is an instruction which is lacking its 'n' address or part of its 'n' address. For example, if the instruction to be output was 'LDN 6 10', the skeleton instruction would probably be 'LDN 6 0'. The addition of '10' to the instruction completes the instruction.

The Instruction Generator (I)

The Instruction Generator uses the integer pair (g,p) to reference a set of linked tables which determine whether an operation is immediately definable - i.e. whether instructions for the complete operation can be compiled - or whether an operation is definable only after doing a mode-auxiliary operation, or whether the operation is illegal. The tables also determine the type of the result operand. If this is a Tx or a Ta operand, the Generator determines its storage address.

There are four sets of tables used in the Instruction Generator.

1. Group Table (QUERY table)
2. Position Table for each Group (CM tables)
3. Instruction Table for each Group (TINS tables)
4. Relativisor Table for each Group (TREL tables)

These tables have the following characteristics.

1. Group Table.

There is only one Group Table. This is referenced by the 'g' integer of the (g,p) pair.

This is a table of two word items, one item per operation group. 'g' gives the address, relative to the start of the Group Table, of the item required. An item gives the addresses of the three tables associated with the group.

The form of a Group Table item is shown below.

1st word.	top 9 bits.	Relative address of the start of the correct TREL table for the group. This is given relative to the start of the first TREL table. All TREL tables are necessarily in successive areas of store.

rest of word. Absolute address of the start of the correct CM table for the group

2nd word. Address of the start of the correct TINS table for the group.

2. Position Table.

Associated with each Group Table item is a Position Table (CMtable). This is referenced by the 'p' of the (g,p) integer pair. Thus any entry in a Position Table is unique to the (g,p) integer pair and can be used to define exactly how an operation is compiled.

An entry in a Position Table consists of one word. If the word is zero, then the required operation is illegal and can not be compiled. A zero entry causes the compiler to flag an error.

If the entry in the Position Table is positive, the required operation can not be completed before a subsidiary operation (inevitably a mode auxiliary operation) has been compiled.

The single word entry is divided into sections as follows.

1. Top bit (bit 0) This determines whether the operation is completed with this compilation
   0 - not completed
   1 - completed.
   (in considering the rest of the sections of this word, if an operation is 'not completed', the other sections refer to the subsidiary operation).

2. Bit 1. This determines whether the PN List will change as a result of this compilation
   0 - no change to PN List
   1 - PN List to change.

3. Bit 2. If the PN List is to change as a result of this operation, this determines which PN item is to be changed.

The local PN list pointer (PNP1) will normally be pointing to the PN item in question. For a sac operation this will not be the top item in the List, but the item to which the ACC marker is pointing. For a Binary operation PNP1 will be pointing to the second top item in the List. For a unary operation, PNP1 will be pointing to the top item in the List.

The item to be changed is determined by this bit as follows

0 - the change is to the item pointed at by PNP1.
1 - The change is to the item immediately above that pointed at by PNP1. In this case PNP1 is changed to point at this item.

(As an example of this, if a Binary operation of the form A ACC op is not immediately definable, but requires a mode auxiliary operation on ACC, PNP1 which would originally have been pointing at 'A' will now be changed to point at ACC becauee ACC is the item in the list which is to be changed).

4. Bits 3-5. For an operation in which the PN List is to change, these bits indicate the type of PN item at the end of this operation. They also indicate whether X3 and/or ACC will be destroyed by the operation.

The bit values and interpretations are as follows.

Value	Result PN item	ACC	X3
0	ACC	destroyed	preserved
1	ACC	destroyed	destroyed
2	X3	preserved	destroyed
3	X3	destroyed	destroyed
4	Ta	preserved	destroyed
5	Ta	destroyed	preserved
6	Tx	destroyed	preserved
7	Tx	preserved	destroyed

If ACC or X3 is destroyed, the associated marker is cleared. However, if the result of an operation is ACC or X3 then the marker will be reset to the PN item for the result.

5. Bits 6-8 These define the number of consecutive TINS table entries which are to be used for this operation. These TINS table entries are generally complete or skeleton instructions.

6. Bits 9-11. For operations where the PN List is changed, these bits give the final mode of the PN item which is changed.

The bits have significance as follows

1. Integer
2. Real
3. Double Precision
4. Complex
5. Logical

7. Bits 13-24. These give the absolute address of the first entry in the TINS table which is to be used for this operation. (Bits 6-8 indicate the number of consecutive TINS table entries which are to be used).

Occasionally it is useful to define an operation which is null (i.e. no instructions are compiled). For example there is a group of operations called 'Load X6' which compile instructions to load a value into X6 from some accumulator. If the original accumulator happend to be X6 then there is no requirement to compile instructions.

A null operation is defined by a CM table entry which is zero except in bits 10-12. These may have any non-zero value.

It is a requirement of the system that all entries to a TINS table from a particular CM table refer to the same TINS table. Further this must be the table specified in the Group Table Entry which specifies the particular CM table.

3. Instruction Table and Relativisor Table.

Associated with each Group Table item is an Instruction Table (TINS table). This is referenced via entries in the CM table which itself is associated with the same Group Table item.

The items in a TINS table are one word long. These may represent one of the following three types of item

1. Complete Instruction
2. Skeleton Instruction
3. C/M

Associated with each TINS table is a Relativisor table (TREL table). This is a character table each entry of which is used to determine the type of item represented by a particular TINS table entry. The character associated with a TINS table entry is the same number of characters distant from the head of the TREL table as the TINS entry is words from the start of the TINS table.

The characters in the TREL table have meaning as shown below.

TREL char.	Associated TINS entry.
0	C/M
1-5	Skeleton Instruction
6-63	Complete Instruction.

If the TREL char. is zero, then the associated TINS entry is taken as a c/m to a store area which contains the name, in characters, of a subroutine. When the Instruction Generator routine references such an item, it ensures that the next time the special relativisor R_4 ('SPRE' within the compiler program sheets) is used in a semi-compiled 5-character group it refers to this subroutine. The subroutine name is also put into the Cue List of the Compiler if it is not already there. 'c' gives the number of words in the name of the subroutine.

If the TREL character is 1, 2, 3, 4 or 5, then the TINS entry is taken as a skeleton instruction. When the Instruction Generator routine references such an item it adds to the skeleton instruction the contents of register NWRD-1 +α where 'α' is the TREL character. This (complete) instruction is then output using the contents of register REL-1 +α as the relativisor. In nearly every case for skeleton instructions, the character is 1 and the registers used are NWRD and REL.

The contents of these registers are defined by the sections of program which process the particular type of operation - e.g. BINARITH for Binary arithmetic operands.

Normally NWRD would be the address in the object program of the non-ACC operand. REL would be the relativisor for that particular address.

If the TREL character is greater than 5, then the TINS entry is taken as a complete instruction. In this case, if the Instruction Generator references such an item it takes the TREL character -1 as the relativisor for the instruction and then outputs the instruction using that relativisor.

As an example of the use of the Instruction Generator, consider the compilation of the operation A+B. A and B are both real variables. A has been assigned location 10 in LV space and B has been assigned 14 in LV space. Both are standard variables.

The PN list contains the addresses assigned to A and B as well as their types (standard variable).

Thus at the time the operation A+B is to be performed, the top of the PN List appears as follows ('var'. indicating 'variable')

Real var./14 (B)
Real var./10 (A)

PNP is pointing at B.

The main (+) operation can not take place until the initial lac operation has been performed. This is done on the variable B in the routine BINSTANDARD using the Instruction Generator.

For this operation, NWRD is set in LOADACC (called from BINSTANDARD) to the address of B (=14) and REL is set to 'VREL' (the relativisor for LV). PNP1, the local PN List pointer is pointing at the item for B.

For the lac operation, 'g' is 12 (See Section on 'Group Identifiers') and 'p' is 2. (2 = REAL variable).

The Instruction Generator is entered with these parameters. The CM table entry for (g,p) = (12,2) is the following (as written in PLAN).

#601/LR+#20000

This indicates that 1. The operation will be completed.
2. The PN List will change.
3. The change is to the item pointed at by PNP1.
4. The result of the operation is ACC and that X3 is not destroyed.
5. The Instruction Generator will reference only one TINS entry.
6. The final mode for the operation will be REAL.
7. The TINS entry is a location labelled in PLAN as LR.

The TREL table entry associated with the TINS table entry is '1'.
The TINS table entry is 'LFP 0'

The Instruction Generator recognizes the TINS entry as a skeleton instruction, and since the TREL entry was 1, adds the contents of NWRD to the instruction. Outputting the instruction using the contents of REL produces the final result.

LFP 14 VREL

The PN List is then changed according to instructions and will now read

Real ACC/- (ACC)
Real var/10 (A)

The operation is now instandard form for a Binary arithmetic operation.

On the next entry to the Instruction Generator, 'g' is zero and 'p' is 56. NWRD is set to the address of A (=10) and REL is set to 'VREL'. PNP1 points at A (the botton item in binary operations).

The CM table entry for (g,p) = (0,56) is the following

#611/ARRR+#20000

This indicates that 1. The operation will be completed.
2. The PN List will change.
3. The change is to the item pointed at by PNP1.
4. The result of the operation is ACC, and X3 may be destroyed.
5. The Instruction Generator will reference only one TINS entry.
6. The final mode for the operation will be REAL.
7. The TINS entry is a location labelled in PLAN as ARRR.

As for the previous case the appropriate TREL table entry is '1'. The TINS table entry is FAD 0 0.

This will produce a final instruction of

FAD 0 10 VREL

Thus A+B would compile into the two instructions

```
LFP     14    VREL
FAD     10    VREL
```

As a further example of the use of the Instruction Generator, consider the compilation of D + I where I is INTEGER and D is DOUBLE PRECISION. I has been assigned location 20 in LV space and D has been assigned 24 in LV space. Both are standard variables.

At the time the operation D+I is to be performed, the top of the PN list appears as follows:-

```
Integer var/20        (I)
D.P. var/24           (D)
```

PNP is pointing at I, The top item in the PN list.
The operation '+' is stored in register OPER.
The main operation (+) cannot take place until the initial lac operation has been performed. For this operation NWRD is set to the address of I (=20) and REL is set to VREL.
PNP1, The local PN list pointer is pointing at the item for B.
For the lac operation, 'g' is 12, and 'p' is 0. (INTEGER)
The CM table entry for (12,0) is the following.

```
#601/LI + #10000.
```

This indicates that the Instruction Generator will reference only one TINS entry, and that the PN list will be changed to ACC at the item indicated by PNP1. ACC will be REAL.
The TREL table entry associated with the TINS table entry is '1'.
The TINS table entry is 'LDX 6 0.'
The Instruction Generator adds NWRD to this entry to form the complete instruction for the operation. This instruction is output using the contents of REL as the relativisor.
The instruction output then is.

```
LDX  6  0   VREL
```

The PN list is then changed according to instructions and will now read:-

```
Integer ACC/-         (ACC)
D.P. Var/24           (D)
```

This is now in standard form for a binary operation.

On the next entry to the Instruction Generator, 'g' is zero and 'p' is 24 (INTEGER ACC, not integer operand). NWRD is set to the address of D and REL is set to 'VREL'. PNP1 points at 'D'.

The CM table entry for (g,p) = (0,24) is the following:-

#301/SBFX + #20000

This indicates that 1. the operation will not be completed, but requires an auxiliary operation.

2. the PN list will change
3. the change is to the top item rather than the one pointed to by PNP1.
4. The result of the operation is ACC; X3 is not destroyed.
5. the final mode for the operation will be REAL.
6. the Instruction Generator will reference only one TINS entry (at location SBFX).

The VREL table entry associated with the TINS entry is #55.
Thus the actual relativisor to be used with this entry is #54.
This is the special relativisor R_1 which will have been set earlier to the name of the Object time Arithmetic Package (%FAP4)
The TINS entry is a complete instruction.

CALL 1 15

which when output with relativisor R_1 is equivalent to

CALL 1 15 %FAP4

(This entry to %FAP4 takes an integer in X6 and generates the equivalent floating point number in the floating point accumulator).
The PN list is now changed according the instructions and will now read.

REAL ACC / - (ACC)
DP. Var/24 (D)

Since the operation was defined as "not complete", the complete operation is attempted again.using the current top items in the PN list.

The operator is still available in ΘPER (+), and the global PN pointer PNP has not been changed since the start of the complete operation and is still pointing at ACC

This time the PN list is in the correct form for a binary operation. On this entry to the Instruction Generator, 'g' is 2 and 'p' is 40 (REAL ACC, Double Precision Operand). NWRD is set to the address of D and REL is set to VREL. PNP1 points at D.

The CM table entry for (g, p) = (2,40) is the following.

#302/DPFX + #30000

This indicates that.

1. the operation will not be completed but requires an auxiliary operation.
2. the PN list will change.
3. the change is to the top item.
4. the result of the operation is ACC; and X3 is not destroyed.
5. the final mode for the operation will be DOUBLE PRECISION
6. the Instruction Generator will reference two consecutive TINS entries (starting at location DPFX).

The REL table entries associated with the TINS entries are both #41. This shows that both TINS entries are complete instructions which will be output using the relativisor #40.

These instructions are:-

```
LDN  4  0
LDN  5  0
```

The PN list is now changed according to instructions and will now read

```
DP ACC / -                    (ACC)
DP Var/24                     (D)
```

Once again, since the operation was defined as "not complete" the complete operation is attempted again.

On the next entry to the Instruction Generator, 'g' is 2 and 'p' is 8 (both operands DOUBLE PRECISION). NWRD is set to the address of D (i.e. 24), and REL is set to VREL. PNP1 points at D.

The CM table entry for (g, p) = (2,8) is the following

#613/ADDR + #30000

This indicates that the operation will be completed, the PN list item pointed at by PNP1 will change to ACC and that ACC will be of mode DOUBLE PRECISION. It further indicates that X3 may be destroyed during the operation.

The Instruction Generator will reference three consecutive TINS entries starting at location ADDR.

The TREL table entry associated with ADDR is 1, the TINS table entry (ADDR) contains

```
LDN  3  0
```

This is a skeleton instruction which is output after adding NWRD. The relativisor is the contents of REL.

Thus the instruction output is

```
LDN  3  24  VREL
```

The second TREL entry is '0'. This indicates that the TINS entry (which is 1/DPAR1) is a c/m to an area which contains the name of a routine . This name is to be output if necessary as a setting for R4 so that the next time SPRE is used as a relativisor (R_4), it will refer to this routine.

The area DPAR1 (one word long) contains the name %FDP.

The third TREL entry is'#60'. This indicates that the corresponding TINS entry (CALL 1 0) is a complete instruction which will be output with the relativisor '#57' (which is R_4)

This then generates CALL 1 0 %FDP

Thus the instructions which are compiled for the complete operation D+I are the following:

```
NOX  6  20  VREL        (I --> X6
CALL 1  15  %FAP4       (Float I)
LDN  4  0               (Convert to D.P.)
LDN  5  0
LDN  3  24  VREL        (D-Address ---> X3
CALL 1  0   %FDP        (I + D)
```

On completion of the whole operation, the Instruction Generator returns control to the Arithmetic Binary routine, which performs certain red-tape operations (such as setting the global PN pointer (PNP) to the local pointer (PNP1))before returning to look for the next operation.

Restriction on the General Use of the Instruction Generator

The Instruction Generator is used throughout the whole compiler. Where it is used from routines other than those which are used for expression evaluation (e.g. from the routines processing the READ and WRITE statements), an operation must be defined to be complete.

When an operation is defined as "not complete" by the Instruction Generator, the return from the Instruction Generator upon generating the auxiliary operation is to a particular location (CMPE) in the Expression Evaluation routine rather than back to the routine from which it was called.

Apart from this restriction, The Instruction Generator is used throughout the compiler in the manner described in the previous section wherever instruction compilation can best be described by 'g, p' values.

Instruction Generator (II)

For many classes of operation, where a group only contains one or two items a different entry to the Instruction Generator (at AC02) is used. For this entry an operation is described by a position number such that no two operation which use this entry have the same position number.

The operations lx3, S<3, and .NOT. are operations which are suitable for this entry to the Instruction Generator. They have position numbers as shown below.

1.	Load X3 Operand is 'S' type	Position Number is 4
2.	Load X3 Operand is T_x type	Position Number is 5
3.	Store X3	Position Number is 6
4.	.NOT.	Position Number is 10

These numbers are used to reference a table (similar to the CM tables) which starts at location AUXCM. The position in the table for an operation determines the position number assigned to the operation.

A table entry is one word long and is a c/m to a second table (identical to a TINS table) which is used for the generation of the actual instructions for the operations.

'c' gives the number of consecutive words in this table which are to be used for the current operation. This may not exceed 7.

'm' gives the address of the first item in this table which is to be used for the operation.

Associated with this table (the AUXINST table) is a relativisor table with the same characteristics as a TREL table. This table is the AUXREL table.

Difference between AC0M and AC02 entry to Instruction Generator

The basic difference between the two entry points is that an operation which uses AC0Z is automatically defined to be complete. This is not the case for entry AC0M. In this case the 'completeness' of an operation is determined by the CM table item for the operation.

A second difference is that an operation defined through entry AC02 does not directly influence the PN list.

If such an operation actually does influence the PN list this is effected externally and not from within The Instruction Generator. Thus although the 1x3 operation will change some PN item from a 'store' or 'Tx' item to an X3 item the actual change to the list is made in the LØADX3 routine which defines the operation, rather than in the Instruction Generator which compiles it.

Operation Compilation (II)

Function Operations.

In the current compiler, a Function reference in an expression consists of evaluating the function arguments and then generating a standard calling sequence

CALL 1 Function Name
argument List.

At the time the CALL is obeyed, none of the arguments may be assumed to be in [illegible] or in X3. The result of a function reference is a value in Acc, the mode being given by the mode assigned to the function 'name'.

Due to the rules for interpreting the PN list, at the time a function operation is to be generated the argument expressions will have been evaluated and the function operator and operands will be in a standard form. This is one of the following

Name a f (one argument function)
Name $a_1 a_2$, a_3 ,---, f (multi argument function)

'Name' is the function name
a and the a_i are the function arguments.

Also, due to the rules for interpreting the PN list one of the arguments may be X3 or Acc

For example, if the Function reference is

F (A(I)) where A is an array name,

this will be in the PN list as shown below at the time the function operation is defined.

F X3 f.

If the function reference had been

F(X+Y)

the PN list at the time the function reference is defined would be

F Acc f.

If one of the arguments is X3 or Acc, a Store auxiliary operation is carried out before the function operation so that in the previous two cases, the end result in the PN list is

1) [illegible] Tx f
2) F Ta f

The operands are now in standard form to be processed for the function operation.

Basically the function operation consists of the generation of the instruction 'CALL 1 Function Name' followed by a list of instructions, one per argument.

The instructions compiled depend on the arguments. For example if the argument is a standard variable, the instruction would be 'LDN 3 Address of Variable' whereas if the argument is a Function

name, the instruction would be 'LDX 3 ADD' where ADD is the address of a constant containing the instruction 'BRN Start of Function'.

The method of recognizing the Function name and generating the associated argument list of instructions is as follows.

1. If the top item in the PN list is not ',' the function has one argument. The second top item in the PN list is the name of the function.

2. If the top item in the PN list is ',' the function has more then one argument. If the list is scanned backwards looking at every second operand (i.e. the third top operand first) the Function name can be found. If the scanned operand is a ',' there are more operands. If the scanned operand is not ',' it is the first argument for the function. The Function name is the next item back in the list.

   The number of arguments is one more than the number of commas found.

At this stage the call instruction can be generated and then the PN list can be scanned forwards starting at the Function name to process the function arguments, ignoring commas until the correct number of arguments has been found.

The resultant PN list item for a function reference replaces the name of the function in the list. This item is Acc and is given the same mode as the mode given to the function name.

Intrinsic Functions

In general, functions allow only a fixed number of arguments. For example SIN is a function written to use one argument and will not work with more than one argument. However there is a small class of functions (called 'Intrinsic' functions) which are defined in the FORTRAN language as allowing a variable number of arguments. For example, MAX1. The names belonging to this class are known to the compiler which marks the function name as belonging to this special class before the name is put in the PN list.

Thus during the analysis of the function operation, the type of the function (whether 'Intrinsic' or not) is known at the time the function name is found.

If the function is an Intrinsic function, the CALL instruction is followed by a word which contains a count of the number of arguments. For example, if the original function reference had been

MAX1(I,J,K,L)

The sequence of instructions compiled would start:-

```
CALL   1   MAX1
           +4          (count of arguments)
```

followed by the argument list.

The complete list of Intrinsic Functions is the following.

AMAX0, AMAX1, MAX0, MAX1, DMAX1.
AMIN0, AMIN1, MIN0, MIN1, DMIN1.

Array Operations

In the current compiler an Array reference in an expression consists

of evaluating the Array subscript expressions and then generating a standard instruction sequence.

1. Load X3 with address of the Array Header
2. CALL 1 %FAP4+6 or CALL 1 %FAP4+20
3. Subscript list.

At 1, the instruction compiled is either

```
        LDN   3   'Address of Header'
or      LDX   3   'Address of Address of Header'
```

depending on whether the Array is not or is a 'dummy' array.

At 2, the CALL is to %FAP4+6 if it is an INTEGER, REAL or LOGICAL array. If it is a COMPLEX or DOUBLEPRECISION Array then the CALL is to %FAP4+20.

At 3. The subscript list is identical to the argument list for functions and consists of instructions to load the address of successive subscripts into X3, one instruction per subscript.

The method of determining the number of subscripts in the PN list and the name of the array is identical to the method used to find the number of arguments and the name for a function.

Function (Array) Argument (Subscript) Lists.

These are identical except that the subscript list only allows constants, variables and expressions. The function list also allows array names and function names.

At the time in the function operation that the PN list is being scanned backwards looking for the Function (or Array) name, the argument (or subscript) list items are being analysed (using a routine ANALYZE). This routine ensures that the argument list will only consist of "LDN 3 α and'LDX 3 β type instructions.

If the argument is an 'unassigned variable' (one which has not been used before in an executable statement or appeared in a COMMON, EQUIVALENCE or DATA statement or appeared as the argument of a FUNCTION or SUBROUTINE statement) the ANALYZE routine assigns space (according to its mode) to the variable and changes the PN list to 'assigned variable'.

If the argument is a workstore (Tx or Ta), the workstore register, CWSA, is set to the value of the argument. e.g. if the argument was the sixth location in LW space, then CWSA would be reset to 6. This would allow the next Tx or Ta item required to be assigned the sixth location in LW space.

If the previous value of CWSA was greater then the 'Maximum Work Store' register (WSMA) then this is set to CWSA before the aforementioned operation takes place.

No change is made to the PN list.

If the argument is a small constant, which normally is a 'Value' operand rather than a 'store operand', a constant of this value is compiled into the LC area and a normal PN item for the argument as a Standard Constant is made to replace the 'Value' item.

If the argument is a function or a Statement Function, a branch to the start of the function or Statement function is compiled into the LC area, and the PN item for the argument is set to look like an indirectly addressed variable (e.g. a COMMON variable) which uses that area.

In this way, the actual instruction compiled for this argument will be LDX 3 'Area', an instruction which, if obeyed, will enable the branch instruction to be passed through for use in the main function for which it is an argument.

If the argument is an array name, the PN item will be changed to indicate "Whole Array as an Argument".

For standard, dummy, DATA or COMMON variables or for constants (other than small integers) the ANALYZE routine does nothing.

Thus at the time of generation of compiled instructions for the argument list, the PN list is such that all arguments can be compiled as 'LDN 'LDN 3 α'or 'LDX 3 β'type instructions.

This is done using a routine LOADX3 which looks at PN list item and compiles the instruction according to the item. Before entry to this routine, PNP1, the local PN indicator is set to point to the item concerned. The routine leaves PNP1 unchanged.

Construction of the PN list in the Compiler

The PN list and operator list in the compiler are contained in a block of 300 words such that the PN list starts at the first location and is generated in a forwards direction, and the operator list starts at the last location and is generated in the reverse direction. Thus the two lists grow towards each other. Each item in each list is one word long. The top of each list is marked by a pointer - PNP for the PN list, and OPTP for the Operator list. Each time an operand is put into the PN list, the value of PNP is incremented by 1. Each time an operator is put into the Operator list OPTP is decremented by 1. The reverse procedure is followed when an operator is removed from the Operator list.

Items are only removed from the PN list when an operation is performed. A local PN list pointer (PNP1) is used during operations so that auxiliary operations can be compiled even though the actual operand for the operator is not the top item in the PN list (e.g. sac, sx3 operations).
At the end of a main operation, the actual PN pointer is set to the current value of the local pointer.

PN List item

A PN list item is one word long and is divided into 3 parts as follows

bits 0-5	Item type
bits 6-8	Item mode
bits 9-23	Item Data

The Item Type indicates whether the PN item represents a Standard Variable, Constant, Temporary Store Location (Ta or Tx), Acc, an array name, or a Function name.
In General it discriminates among all possible operand types which can be distinguished either by their use or by their type of storage area. For example a variable defined in a DATA statement will have a different 'Type' to a normal variable. A variable which was defined as an argument in a FUNCTION or SUBROUTINE statement will again have a different Item type. Ta and Tx operands, although they occupy the same type of storage have different Type bits because they are not both used in the same manner. A small integer has a different Type to any other constant because it does not normally exist in a store location.

The item mode gives The mode of the operand as follows

1 - INTEGER
2 - REAL
3 - DOUBLE PRECISION
4 - COMPLEX
5 - LOGICAL

The Item Data usually gives the address in the object program for the item in question. This address is given relative to the start of the local storage area.
For example if the item was a Standard Variable which had been assigned a block of store starting at location 10 in Lower Variable space, the Item Data for that variable would be 10.

For small integers, the Item Data gives the value of the constant.

For certain operands (e.g. Function names and undefined items), the Item Data gives the address in the Compiler Main List of the list item for that operand.

The Item Data for the ACC operand is undefined.
A 'comma' is put into the PN list as a zero word.

The following table gives the Type codes and Data in the PN list for all operands.

Quantity	Type Code	Data	Rel	Dor I
Absolute Store	0	Address of Store	NORE	D
Standard Variable	1	Address of Variable	VREL	D
Constant	2	Address of Constant	CREL	D
Small Integer	3	Value of Integer	NORE	D
Common Variable	4	Address of Constant referencing var.	CREL	I
Dummy Variable	5	Address of location referencing var.	WSRE	I
Stat. fcn. argument	6	Address of location referencing arg.	WSRE	I
Dummy Array Name	7	Address of location referencing Header of actual Array	WSRE	I
Dummy Function Name	8	Address of location branching to actual Function	WSRE	I
Function Name	9	Address of List Item for Function	SPRE	I
Stat.fcn. Name	10	Address of start of Stat. fcn.	P2RE	I
Work Store (Ta)	11	Address of Store	WSRE	D
Normal Array Name	12	Address of Array Header	CREL	I
Work Store (Tx)	13	Address of Store	WSRE	I
Total Array	14	Address of Array Header	CREL	D
Dummy Array	15	Address of word referencing Header of Actual Array	WSRE	I
Intrinsic Function Name	16	Address of List Item for Function	SPRE	I
DATA variable	17	Address of Variable	CREL	D
X3 = address of result (X3)	18	--	NORE	D
Unknown	20	Address of List Item.	--	-
Stat. fcn. definition	21	--	--	-
Accumulator (Acc)	32	--	--	-

The column 'Rel' in the table indicates which relativisor is to be used if the operand is to be used in a skeleton instruction. Thus, if the operand is a Standard Variable, it would appear with the Item Data in the modifier part of the instruction which would be output using the relativisor 'VREL'.

The column D or I indicates whether a 'Load' operation (load X3 or Acc) is to assume the item in lower storage. If 'D', the item is in lower storage. If 'I', the item must be considered as though it is in upper storage.

These two characteristics of the operands ('Rel' and 'D or I') are found by referencing a character table which contains the relativisors and a 'bit' table which gives the D and I information.

Thus from these tables, all the information necessary to define an operand can be found.

Complex Constants

A complex constant has the form

'(a,b)' where a and b are optionally signed REAL constants. This construct must effectively give rise to one PN list item for the constant and no Operator list items.

This is effected as follows:

1. The '(' goes into the Operator list
2. The sign, if there, goes into the Operator list.
3. The constant 'a' is recognized as a REAL constant.
   The constant terminator is recognized as ','

Only at this point is the construct recognized as that of a complex constant. Further, the construct will only be recognized as such if it is not immediately preceded by an Alphanumeric or numeric field.

e.g. F(+3.1, - 2.4) defines a function reference
whereas +(3.1, - 2.4) defines a complex constant.

At this stage, the top operator is removed from the operator list. If it is 'unary minus', the constant 'a' is negated and stored in a temporary store. The '(' is also removed from the operator list.

'b' is then found. If this is signed and if the sign is 'unary minus', 'b' is negated and stored in NAME+2, NAME+3. The original Temporary Store is transferred to NAME, NAME+1 giving a block of four words containing the internal representation for the complex constant.

The final ')' is checked and then ignored.

The constant 'terminator' is then found. (The operator separator or terminator following ')').

The whole construct has now been transformed to the form of a constant in NAME→NAME+3, and a terminator in TERM, which is the standard form for the analysis of any operand-operator pair.

Standard and Intrinsic Functions

Most functions, if not defined in 'Type' statements are assigned a mode INTEGER or REAL according to the first letter of the function name. However, there is a small class of functions which do not follow the normal rule. These are standard functions which are defined to have a Double Precision COMPLEX or LOGICAL mode.

There is a second class of function, which has the characteristic that each member of the class allows a variable member of arguments. This is the class of 'Intrinsic' functions. This second class contains some members of the first class.

The two classes of function names form a closed set, all entries of which are known to the compiler.

Whenever the compiler processes a function name found in an expression for the first time, it checks whether the name belongs to this set. If it does, the Main List Item for the name is updated to contain the characteristics for the name as defined in the set (i.e. mode and type of function - standard or intrinsic) unless the List Item as already made has had a mode assigned to it (by a 'Type' statement) which is different from the mode defined in the set. If the mode is different, the function is considered as similar to a Standard Function even if it has the same name as an Intrinsic function.

Operator List

The operator list consists of one word items each of which represents an operator or a separator.

The design of the operator item is shown on the next page

1. Top 21 bits of the word. These indicate the 'weight' of the operator. This varies from -1 (#7777777) for ')' to 9 for the Function and Array pseudo-operators (There is another pseudo operator which has not been mentioned with a weight of 10.)

2. Bottom 3 bits. These are used to distinguish among operators of the same weight.

The complete list of operator items is given below

Item	Representation	Item	Representation
)	#7777777 7	.GT.	#4 4
(	#0 0	.GE.	#4 5
=	#0 2	+	#5 0
,	#0 4	-	#5 4
.OR.	#1 0	$-_u$	#6 0
.AND.	#2 0	*	#7 0
.NOT.	#3 0	/	#7 4
.LT.	#4 0	**	#10 0
.LE.	#4 1	a	#11 0
.EQ.	#4 2	f	#11 2
.NE.	#4 3	ifq	#12 0

The operator ifq is a special pseudo-operator which is used when the operation type is not defined but may later be found to be one of

1. Statement Function definition
2. IF statement
3. Array operation
4. Function operation.

In this table the 'weight' is used to reference a separate character table to define the class of operation involved (e.g. Arithmetic Binary, Unary Minus etc.)

V.K. Taylor