THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

THE DESIGN AND IMPLEMENTATION OF A LANGUAGE FOR MANIPULATING ALGEBRAIC FORMULAE

J. M. OFFICER

Presented for the Degree of Doctor of Philosophy

at the University of Edinburgh

1971

ACKNOWLEDGEMENT

I would like to thank my supervisor, Professor Michaelson, for all the help he has given me while I was engaged on this project.

Contents

I Introduction

II Facilities Required in Manipulating Algebraic Expressions

III The Examination of Some Existing Languages

IV Some Design Features of an Algebraic Manipulation Language

V A Full Description of AML

VI Implementing AML - The Storage Tree

VII Implementing AML - Analysis of a Statement

VIII Implementing AML - Organisation of Store

IX Implementing AML - Garbage Collection

X Implementing AML - Polish Notation and Expressions

XI Implementing AML - The Commands

XII Implementing AML - The Algebraic Commands

XIII Implementing AML - Patterns

XIV Conclusion

THIS THESIS EXPLORES THE POSSIBILITIES OF DOING MATHEMATICAL PROBLEMS INVOLVING ALGEBRA ON A COMPUTER. A LANGUAGE IS DESIGNED WHICH ALLOWS NAMES TO OCCUR AS UNKNOWN QUANTITIES. THIS LANGUAGE HAS ALL THE FACILITIES OF A GENERAL PURPOSE LANGUAGE SUCH AS IMP, BUT IS DESIGNED TO BE USED INTER-ACTIVELY BY A USER AT A CONSOLE. THE LANGUAGE ALSO INCLUDES INSTRUCTIONS WHICH CAUSE THE USUAL ALGEBRAIC OPERATIONS TO BE APPLIED TO EXPRESSIONS. THESE OPERATORS INCLUDE SIMPLIFICATION, DIFFERENTIATION, BUT NOT INTEGRATION.

A BRIEF SURVEY IS GIVEN OF OTHER LANGUAGES IN THE FIELD, WITH COMMENTS ON THEIR CAPABILITIES AND RESTRICTIONS.

THE SECOND PART OF THE THESIS DESCRIBES HOW THE LANGUAGE IS IMPLEMENTED. AN INTERPRETER IS USED. STATEMENTS OF THE LANGUAGE ARE ANALYSED SYNTACTICALLY AND THEN OBEYED. ALGEBRAIC EXPRESSIONS ARE STORED IN BYTE ARRAYS, USING A TYPE OF PREFIX POLISH NOTATION.

FINALLY THE LANGUAGE IS REVIEWED IN THE LIGHT OF RECENT WORK DONE IN THE FIELD, AND SUGGESTIONS ARE MADE FOR A FURTHER VERSION.

I Introduction.

Since the development of FORTRAN by I.B.M. in 1957, numerous programming languages have been designed. Many of these, however, have been discarded. The two now most commonly used are FORTRAN and ALGOL60. These are sometimes referred to as 'algebraic languages' because a name may be used to represent a location in store, and arithmetic operations may be performed on such names. An instruction in these languages looks like an algebraic formula.

E.g.

A = B + C

However, in one important respect, this terminology is misleading. The instruction means: 'Find the number in location B, add it to the number in location C, and put the result in location A.' That is to say at the time the instruction is carried out, it is necessary for B and C to represent actual values: there is no provision for unknown quantities.

As more and more people began to use these languages, it became apparent that there was a need for some means of doing truly algebraic work by computer. In many branches of science people require to manipulate large expressions containing unknown quantities. The work is tedious and error prone.

It is not, though, entirely mechanical. Consider the expression

a * x + b * (x + y) + a * y.

If a, b, x and y are given values, this expression has a unique value, obtained by performing the operations indicated. However if the four names are unknown quantities, then there is no unique way of representing it. Should the expression be considered as it is, or in the form

a * x + b * x + b * y + a * y

or in the form

(a + b) * (x + y)?

There is no correct answer to such a problem. The best form for an algebraic expression to take depends entirely on the problem that is being solved.

Despite this difficulty, many attempts have been made to write languages, programs and packages to enable users to manipulate algebraic expressions. Indeed there was a sufficient number for Jean Sammet of I.B.M. to produce in 1966 a bibliography of the languages written up to that time ([1] pages 555 - 569).

In writing yet another language, one implies that those already written are not satisfactory. Therefore we must first examine the requirements of a user wishing to do algebraic manipulation by computer, and then examine the existing languages to decide whether any of these satisfy the requirements.

II Facilities Required in Manipulating Algebraic Expressions

We have indicated that there are problems in the field of algebra which are sufficiently long and tedious, and to a certain extent mechanical, to merit trying to solve them by computer. However in solving algebraic problems we also rely quite heavily on our ability to 'spot' certain equalities and relations where a general description of the connection between the two expressions would not be easy. The same intuitive thinking is used in deciding which form of an expression is most suitable for a particular problem. (As discussed in Chapter I.) Therefore, although a computer by itself could be used to alleviate some of the burden, the combination of the machine's power and man's intuition and large store of knowledge would seem the best choice.

Luckily, recent developments in the study of computer systems have shown a way to provide an ordinary user with the ability to communicate directly with the computer. Such systems are known as multi-access, or in some places, time-sharing systems. Each user is provided with a teletype or some other kind of terminal and can type instructions to the computer, and obtain its replies immediately on the same sheet on which he printed his instructions. This method of communicating with a computer is said to be on-line or interactive.

This requirement, that an on-line system be used, affects the other decisions taken about the facilities required. If a user can look at the result of one instruction before issuing the next, he does not have to program instructions which take care of cases which have not arisen (but which might have done). However if the processing of a particular job is standard, he will not wish to supervise it. Hence we require an interactive language which is capable of running on its own.

Although a problem may require algebraic manipulation, this does not preclude the desirablity of normal numerical programming facilities. In fact all the power of a general purpose language should be available to the user, preferably without any need to convert the algebraic expressions into some other form before this power can be used.

We have talked about manipulating algebraic formulae so far without specifying what this involves. We shall now consider some of the operations that will be required.

1) Simplification.

In the example given in Chapter I, none of the three forms mentioned can be regarded as simpler than the other. However there are cases where one form would generally be accepted as simpler than another. For instance, one would say that the expression 'a' is simpler than the expression 'a+0'. If we only allow very obvious rules such as this, too much work will have to be done manually for the language to be of much

practical value. Consequently we must devise a set of rules that define simplification in the language, and use these. They will not suit all cases, and so a way must be found to do more, or less, simplification by means other than the simplification instruction that is provided.

The following rules could define the simplification instruction.

(i) Assume the operations '+' and '*' are commutative and associative.

(ii) If a is an unknown variable,

```
a+0 -> a
a*0 -> 0
a*1 -> a
a**0 -> 1 ('**' denotes exponentation.)
a**1 -> a
```

(iii) If m and n have numerical values,

```
a*a -> a**2
a**m*a**n -> a**N where N = m+n
a+a -> 2*a
m*a+n*a -> N*a
a**m**n -> a**M    where M = m*n
m*a/(n*a) -> P*a   where P = m/n
m*a-n*a -> Q*a     where Q = m-n
```

(iv) If a term is defined as a series of unknown variables that are multiplied together, then identical terms are simplified according to the

rules given in (iii) for a single unknown variable.

Hence

a*b+a*b -> 2*a*b

m*a*b+n*a*b -> N*a*b

(a*b)**m*(a*b)**n -> (a*b)**N

etc.

(v) Rule (iii) is applied to the variables of a term before it is considered as a whole. For example

a*a*b -> a**2*b.

(vi) From rule (v) we obtain an extension of the definition of a term, i.e. that it is a series of unknown variables, each possibly raised to a power, that are multiplied together. From (iv) and (v) we get

a*a*b + a**2*b -> 2*a**2*b

etc.

(vii) The commutative law of multiplication will allow us to recognise 'a*b' as equal to 'b*a' and to apply the rules of (iv). Hence

m*a*b + n*b*a -> N*a*b

etc.

(viii) The associative law will allow us to extend (vii) to terms consisting of the same variables, but listed in different orders. For instance

m*a*b*c*d + n*b*a*d*c + p*b*c*d*a -> S*a*b*c*d

where p has a numerical value, and

S = m+n+p

(ix) We can define an expression as a series of terms added together. Then expressions which are identical and which occur as sub-expressions of a larger expression can be treated in the same way as a single unknown variable, and the rules of (iii) are applied to the large expression.

E.g.

m*(a+b) + n*(a+b) -> N*(a+b)

(x) Sub-expressions whose terms are equal can also be simplified in accordance with the rules of (iii).

E.g.

(a*b+c)**m * (b*a+c)**n -> (a*b+c)**N

(xi) The commutative and associative laws for addition will allow us to extend (x) to apply to any equal subexpression.

E.g.

m*(a*b+c) - n*(c+b*a) -> Q*(a*b+c)

2) Substitution.

The ability to substitute names or expressions in an expression will give the user control over the exact format of the expression. Hence, referring back to the discussion on simplification, if the rules provided do more than required in a particular case, one could cause those that were required to be carried out by substitution. For example; substitute 'a' for

'a+0'. Also, the user may use this facility to replace one form of an expression with another, where there is no facility to do this automatically. The many trigonometric identities could be used in this way.

3) The Distributive Law and Factorisation.

One cannot say that the expression 'a*(b+c)' is simpler than 'a*b+a*c', or vice versa. Therefore it is suggested that the conversion of one of these forms is not done by the simplification process, but that other facilities are provided. The transformation from 'a*(b+c)' to 'a*b+a*c' is quite mechanical, but its opposite presents more problems. There is a unique way of factorising 'a*b+a*c', but for other examples this is not the case.

a*b+a*c+d*c

could be factorised as

a*(b+c)+d*c

but also as

a*b+(a+d)*c

Therefore a facility more simple than factorisation should be supplied, and one that gives a unique result. Collecting terms which multiply a given expression should be sufficient for our needs. Factorisation would be obtained by repeated calls on the 'collect terms' facility.

For example,

a*c + a*d + b*c +b*d

would be factorised by calling collect-terms for a, and then collect-terms for b. The first call would give the result

b*c + b*d + a*(c+d),

and the second,

b*(c+d) + a*(c+d).

Finally we call collect-terms for (c+d), to give

(b+a)*(c+d).

4) Differentiation and Integration.

Differentiation is a mechanical process which readily lends itself to computerisation. Integration presents quite a different problem. Some standard integrals can be performed mechanically, but a large number of problems can only be solved by trial and error methods. Hence it seems that a limited integration facility could be supplied, with the capability of returning partial results for the user to examine.

Having discussed some of the manipulations that could be applied to algebraic expressions, we must decide what form the request for a manipulation would take. Two methods suggest themselves. The name of the facility could be built into the language, or they could be used as routine or function calls. Both have disadvantages. The former requires an alteration to the program implementing the language, every time there is a need for a new or modified facility, while the latter is difficult to read. On the grounds that

the user should be given a clear and readable language, and should suffer as little as possible from the difficulties encountered in the implementation, I would choose the former method. No language, though, should be severely criticised for choosing the other method.

Most general purpose languages offer the user integer or real arithmetic. However in working with algebraic expressions, one finds that the use of a rational number often shows up some quality more clearly than the equivalent real number would. Hence rational arithmetic should be available, and also a mechanism for handling complex numbers. The range of the rationals should be as large as possible.

A form of algebraic expression which should be given special consideration is the polynomial. This is used so frequently in scientific problems that facilities must be available for manipulating it. In particular infinite polynomials must have a representation, There should be a facility for handling general infinite series too.

The user of a language that manipulates algebraic expressions requires to supervise very closely the expressions he is considering. He may wish to examine and alter only part of an expression. Hence he must be able to specify, replace or delete this part. Also he may not be interested in the exact expression, but only its general form. For instance, if an expression is of the form ′something**2 - somethingelse**2′, he may want to replace

this by the product of the sum and difference of the two terms, no matter what ´something´ and ´somethingelse´ are. Hence there must be a way of matching a specific expression against a general form.

Throughout this Chapter we have been discussing expressions. However we also wish to examine algebraic equations, moving terms from one side or another, or solving for a particular variable. Let us call the set of equations and expressions together formulae.

We have also assumed that the formulae will belong to a field. Provision should be made also for non-commutative and non-associative algebra. The use of vectors and vector algebra could also be considered.

III The Examination of Some Existing Languages.

In Chapter 2 we discussed the desirable qualities of a language for manipulating algebraic formulae. We need an interactive language which has all the power and facilities of existing general purpose languages. In addition, we require a set of algebraic operators to be applied to algebraic formulae, rational arithmetic, a representation of infinite sums (and products), and some kind of pattern matching facility. Now we shall discuss briefly some of the existing languages for manipulating algebraic formulae.

This chapter is divided into two parts; the first is a description of the languages that were available when work on AML was begun. The second part describes some of the languages available today. The conclusion of this chapter shows that there was a real need for an improved language at that time; a comparison of AML with some of the languages that have been written since is given in Chapter 14.

3.1 Languages Available in 1967.

3.1.1 Van de Reit´s Package

In 1966, Van de Reit described a package of ALGOL60 routines which could be used for algebraic manipulations. (See [2], pages 64 - 70, and [3].) Since he was working within the limits of a general purpose language, he could not represent an algebraic expression in its natural form. Hence for example, he defined ´+´ and ´*´ as functions

having two arguments. The sum ´a+b´ would be represented as S(a,b), and ´a*b´ as P(a,b).

Two systems have been developed - a simple one and a general one. Van de Reit describes the simple one in detail and then gives the additional features of the general system. Both are sets of ALGOL procedures.

a) The Simple System.

This permits two operators, + and *, and can differentiate and output expresssions. These are represented by the procedures

S (sum) for +

P (product) for *

DER for the derivative

OUTPUT for outputting formulae.

The formulae are stored internally in a 2-dimensional array F. If the formula is an algebraic variable, the first and last entries of F are zero, and the middle entry is 3, as shown in Fig. 1.

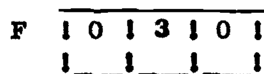


Fig. 1

If the formula is the sum of two terms, the first entry points to the first operand, the middle entry has the value 1, and the last entry points to the second operand. The product of two terms is represented in a similar way, but the middle entry is 2.

The Simple System can be divided into two parts - the outer block contains all the standard routines, and also initialises several variables. The inner block contains the user's program.

One and zero are special algebraic variables which have properties that are recognised by the system. An example given for the simple system is

```
1       x:=STORE(0,algebraic variable,0);
2       y:=STORE(0,algebraic variable,0);
3       f:=S(x,y);OUTPUT(f);
4       f:=P(x,y);OUTPUT(f);
5       f:=P(S(x,y),S(x,y));OUTPUT(f);
6       f:=DER(f,x);OUTPUT(f);
7       f:=DER(f,y);OUTPUT(f);
```

This is the program that the user must write. The statements mean:

```
1&2     x and y are algebraic variables.
3       f = x+y, print out f.
4       f = x*y, print out f.
5       f = (x+y)**2, print out f.
6       f = df, print out f.
            --
            dx
7       f = df, print out f.
            --
            dy
```

The situation after the first two statements have been obeyed will be

```
              F
          _____________
one  ->-  ! 0 ! 3 ! 0 !
zero ->-  ! 0 ! 3 ! 0 !
 x   ->-  ! 0 ! 3 ! 0 !
 y   ->-  ! 0 ! 3 ! 0 !
          !___!___!___!
```

Fig. 2

S (line 3) finds the sum of its two parameters. If either of them equals zero, the result of S is a pointer to the other. Otherwise STORE is called to make a new entry in F, as shown in Fig. 3.

```
              F
          _____________
one  ->-  ! 0 ! 3 ! 0 !   1
zero ->-  ! 0 ! 3 ! 0 !   2
 x   ->-  ! 0 ! 3 ! 0 !   3
 y   ->-  ! 0 ! 3 ! 0 !   4
 f   - -  ! 3 ! 1 ! 4 !   5
          !___!___!___!
```

Fig. 3

The other routines proceed in a similar fashion.

b) The General System.

The internal representation described in the Simple System is also used in the General System. The facilities offered in the General System, in addition to those of the Simple System are

1) D(x,y) = x-y

2) Q(x,y) = x/y

3) POWER(x,y) = x**y (i.e. x raised to the power of y)

4) INT POW(x,i) = x**i, where i is an integer expression.

5) Complex, real or integer numbers may be used in formulae.

6) POL(i,d,f,c) which is a polynomial of degree d, of the form

$$\sum_{i=1}^{d} c(i)f{**}i$$

7) Special functions: EXP, LN, SIN, COS, ARCTAN, SQRT.

8) Sum(i,j,k,f) which is

$$\sum_{i=j}^{k} f(i)$$

9) SIMPLIFY(f) - simplify the formula.

10) CC - complex conjugate.

11) SUBSTITUTE

12) QUOTIENT(f,g,r) - Divide f by g, putting the remainder in r.

13) COMMON DIVISOR(f,g)

14) RN - evaluate a real number.

15) IN - evaluate an integer.

16) CN - evaluate a complex number

An example is given in [2] which calculates eight Taylor coefficients of sin(x).

c) Organisation of Space.

In the Simple System, F is set to be 1000 long. In the general system, space is reserved by making F an own array in the procedure INT REPR which stores

formulae internally. F is made to grow whenever more space is needed, by declaring it as

```
OWN INTEGER ARRAY F(1:kmax,1:3)
```

Kmax is increased if more space is needed. Note that this method would not be possible in IMP, which allows only constant bounds to an own array.

d) Notes

1) A boolean variable expand is used. If it is TRUE, all formulae are expanded, i.e. have the distributive and other laws applied to them. If it is FALSE, the formulae are stored as presented.

2) Deleting formulae. Three routines are available: FIX, ERASE, and LOWER INDEX. These have the following meanings.

   ERASE - remove all formulae down to the last FIX.

   FIX - protects formula used before it.

   LOWER INDEX - removes the effect of the last FIX. (ERASE also does this.)

3) Polynomials are treated as truncated power series. A maximum degree for all polynomials is set when the program begins. Polynomials may be added, multiplied or divided.

4) Simplification - f and g are equal if

   (i) f and g are numbers, and f=g.

   (ii) f and g are the same algebraic variable.

   (iii) f - g = 0 when simplified.

(iv) If f and g are equal, and n1, n2 are numbers, then n1*f+n2*g is simplified to (n1+n2)*f.

5) COMMON DIVISOR and QUOTIENT have proofs of termination.

e) Conclusion

The author claims that this system has advantages over others like FORMAC and Formula Algol, since it does not require any facility other than an ALGOL compiler. Alterations to the system are easier since they do not involve machine code. He admits that it takes up time and space, but without giving any figures.

Offset against the advantages of writing ALGOL procedures must be the unwieldiness of the actual program. In order to alter the procedures, the user must have a good knowledge of the system. Therefore this advantage only applies to specialists who are prepared to take the time and study it.

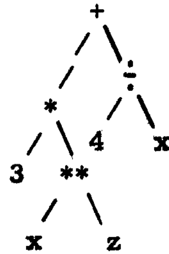
3.1.2 Formula Algol.

This is an extension of ALGOL which incorporates formula manipulation and list processing facilities. The latter will not be described here. It was developed by A.J.Perlis at Carnegie Mellon University [4].

A new type called FORM is introduced. If a variable is declared to be of this type, any expression assigned to it is stored as a binary tree. For example,

```
3*X**Z + 4/X
```

is stored as



The value of a FORM variable is built up of atomic formulae and numeric constants. When the variable F is declared to be of type FORM, its value is set to the atomic formula f (i.e. it has its name as its value). FORM arrays may be declared, but their values are not initialised.

When an assignment is made, the value of the operands is substituted, even for a FORM variable.

Example

If I is an integer with the value 2, and F and G are FORM variables, G having the atomic formula g as its value, then

```
F := I + G;
```

gives F the value 2+g.

```
F := F**2
```

then gives F the value (2+g)**2. All operators are binary. Thus

```
F := G + I +1;
```

will give F the value g+2+1, but

```
F := G +(I +1);
```

gives F the value g+3, since the subtree in the second case

can be evaluated.

The assignment of the value of a FORM variable can be prevented by preceding the variable name by a dot. Thus

```
H := .F + 1;
```

will give H the value F+1, and not g+1+3.

Conditional formulae, procedure calls and array accesses can also be delayed. Examples are

```
H := .IF I=1 THEN X ELSE Y;
H := P.(A1,A2);
H := A.(1,4);
```

where P is a procedure and A an array.

The delay in evaluating caused by the dot operator can be removed by the operator EVAL, as in the following example.

```
F := G + 3; (where G is atomic)
H := .F +1;
F := F*3; (value is (g+3)*3)
EVAL H;
```

which gives H the value (g+3)*3+1. EVAL may also be used to substitute new values in an expression. So

```
EVAL (1,Z+1) F (X,Y);
```

means substitute 1 for X and Z+1 for Y in F. Hence if

```
F := X+Y;
```

the result of this EVAL will be 1+z+1.

EVAL also carries out some trivial simplification, e.g. a*0 is replaced by 0.

Another operator SUBS does only substitution, in the manner described for EVAL. REPLACE(F) finds the current values of any FORM variables in a formula F, and substitutes these. For example, let

```
          F := X + Y * Z;
          Y := 1 ; Z := 2;
then      SUBS (3,4)F(Y,Z);
```

gives F the value x+12, but

```
          REPLACE(F);
```

gives F the value x+2.

Patterns.

The real power of Formula Algol lies in its pattern matching facilities. These are used to determine the structure of a formula. There are two Formula Patterns, each of which are regarded as Boolean expressions.

1) FORMULA == PATTERN

2) FORMULA >> PATTERN.

The first gives a result TRUE if the FORMULA is an instance of the PATTERN. The second gives the result TRUE if the FORMULA contains an instance of the PATTERN.

F == P can be defined recursively as follows.

1) If P is an atomic formula, the expression is TRUE if and only if F is that same atomic formula.

2) If P is the type name REAL, INTEGER, BOOLEAN, FORM, the expression is TRUE if and only if F is

real number, an integer, a logical value, or a formula, respectively.

3) If P is the reserved word ATOM, the expression is TRUE if and only if F is an atomic formula.

4) If P is the reserved word ANY, the expression is always TRUE.

5) If P = QwR, where Q and R are patterns and w is an operator, then the expression is TRUE if and only if F =GvH, where G==Q, H==R and v=w.

Operator Classes. This facility enables the user to define sets of operators, any of which will satisfy his needs. To do this, another type, SYMBOL must be used. If A is of type SYMBOL, then the statement to define an operator class is as follows.

A <- /[OPERATOR: +,-][COMM: TRUE,FALSE][INDEX j].

Then if P = Q!A!R, F == P is true if F = GwH and one of the following is true.

1) G == Q, H == R and w='-'

2) G == Q, H == R and w='+', or

3) G == R, H == Q and w='+'

Thus it is possible here to test for the operators + and -, taking into account the commutativity of +. The index j is set to 1 if w='+', and to 2 if w='-'. Once j has been set, the test

F == Q!<A>!R

will look for the operator defined by j.

F >> P is true if there is a sub-formula, S, of F such

that S == P.

Extractors may be used to show how the formula was split up during the pattern match. Thus if F= x+y*z,

    F == Q:ANY+B:ANY

(where Q and B are the extractors), will result in Q having the value x, and B the value y*z.

Also, the test A:F>>B:P may be used. This has the following result. If F >> P, B has as its value the part of F that P was an instance of. A has as its value F, with the part that B points to replaced by the previous value of B.

For example, let

    B = 5 + x

    F = x + y + z

then A:F >> B:ATOM

will result in B having the value x, and A the value

    5 + x + y + z.

Transformed Formulae.

If F and G are formulae, and if P is a pattern, a PRODUCTION P -> G, may be said to be applied to F. In this case, the following takes place. The test F == P is made, and if it succeeds, F is changed, according to rules given by G. An example best illustrates this. Consider the production

    A:ANY*(B:ANY+C:ANY) -> .A*.B + .A*.C   (i)

If this is applied to the formula

    X**2 * (Y + SIN(Z)),

the test F == P succeeds, and the extractors have values as follows

A has the expression X**2

B has the expression Y

C has the expression SIN(Z).

Then replacing F by the structure defined in (i) gives

F = X**2*Y + X**2*SIN(Z).

A schema is a list of such productions. A schema is applied to F by the statement F ↓ S (↓ is a downward pointing arrow in the original documentation.) S may have two forms

S <- [P1, P2, ... Pn]   (a)

S <- [[P1, P2, ...Pn]]   (b).

where P1 ... Pn are productions. The method of application of these productions differ.

a) One by one sequencing

The Pi´s are applied to F until a success is obtained. The transformation defined by that Pi is applied to F, and the process begins again at P1. If a Pi fails, it is applied to each sub expression of F before Pi+1 is tested.

b) Parallel sequencing.

The Pi´s are examined as before, and the process returns to P1 if a successful transformation is found. However, the Pi´s are applied only to the top level of F in the first instance. If all the Pi´s fail at the top level, they are applied to

the subexpressions of F at the next level, in a recursive fashion.

Apart from SUBS and EVAL, which do very elementary simplification, Formula Algol has no built in algebraic operators. The pattern matching facilities and the schema allow the user to build up his own rules.

3.1.3 FORMAC

FORMAC, ([5], pages 474 - 491, 6 , and [7] pages 37 - 53) was developed by I.B.M. in 1964, under the direction of Jean Sammet. Two versions are available; the first was designed for use with FORTRAN IV, and the second to be used with PL/1. The Edinburgh Regional Computing Centre uses the second version with FORTRAN IV G.

Both have the same basic philosophy: FORMAC statements may be interspersed with statements of the host language. A pre-processor is applied to a FORMAC program to translate it into a program in PL/1 or FORTRAN. The details of the two versions differ, and the second will be described here; because it is the later version, and also the one available at Edinburgh.

FORMAC statements are distinguishable from FORTRAN statements as they are preceded by the word LET, and placed in parentheses. Thus

```
X = 1
```

is a FORTRAN statement.

```
LET ( X = A + B)
```

is a FORMAC statement.

Unless they have already been assigned a value, FORMAC variables are ATOMIC, i.e. they are treated as algebraic constants. Thus if the FORMAC variable B had been given the value, 3, but A had no value, X would contain the value A+3. FORTRAN and FORMAC variables are quite seperate; if a FORTRAN variable is used inside a FORMAC statement, it is enclosed in ?'s, e.g. ?x?.

Rational arithemetic is assumed in FORMAC statements, unless a real number occurs in the expression.

Whenever an expression is assigned to a FORMAC variable, it is simplified automatically. This automatic simplification is a deliberate design feature. The reasons for choosing to do this appear to be twofold.

1) The authors maintain that users will, in general, require simplification.

2) Storing expressions in a simplified form saves space. They found that restrictions on FORMAC programs was generally caused by lack of space, and not time.

Although simplification is automatic, there are three commands which are explicit.

MULT	Apply the multinomial law.
DIST	Apply the distributive law.
EXPAND	Apply both the multinomial and the distributive laws.

In addition, there are two substitution commands

EVAL(expr,a1,b1,...an,bn)

Replace ai by bi in expr, in parallel, for each ai.

REPLACE(expr,a1,b1,...an,bn)

Replace each ai by bi in expr, sequentially.

The difference between these two is best illustrated by an example.

EVAL(A+B+C,A,B,B,C,C,A)

gives B + C + A.

REPLACE(A+B+C,A,B,B,C,C,A)

gives B+B+C, replacing A by B.

C+C+C, replacing B by C for all C's.

A+A+A,(= 3*A), finally.

FORTRAN functions, such as SIN, COS etc may take symbolic expressions in FORMAC statements. There are also additional functions.

FAC factorial

COMB(n1,...nk) = n1(n1-1)....(n1-n2-..nk+1) / n1!....nk!

If these functions are used in FORMAC statements, options may be set to indicate whether they are to be evaluated. For example,

LET(A = SIN(3/4))

If the option TRANS is on (set by the command

OPSET(TRANS))

then A gets the value 0.68163876. If it is off, (set by OPSET(NOTRANS)), the value of A is SIN(3/4).

There is also a User defined function form

FNC(f) = expr($(1), $(2), ... $(n))

where the $(i) are formal parameters. For example,

```
FNC(ROOT) =
(-$(2) + SQRT($(2)*$(2) - 4*$(1)*$(3))/(2*$(1))
```

Then

```
Y = ROOT(A,2,3)
```

gives Y the value

```
-2 + SQRT(4 - 12*A))/2*A.
```

FORMAC expressions are stored internally in Delimiter Polish Form. This is a form of prefix Polish notation which treats + and * as n-ary operators. The scope of these operators is defined by a delimiter ´]´ after the last operand. Thus

```
A + B + C + D
```

is represented as

```
+ A B C D]
```

and

```
A + B*5 + C + D is
+ A * B 5] C D].
```

In addition, these are put in a canonical order, determined by the alphabetical order of the variable. This is done prior to simplification.

Both FORMAC and Formula Algol are extensions of widely used programming languages designed to be run in batch mode, although there is a desk calculator version of FORMAC. It is worth quoting Jean Sammet´s comment in her survey of formula manipulation languages (see [1], page 253.)

> ´Formula Algol provides a general basic manipulation framework from which a user can build

> up just about anything he wants to do, as with an assembly program; this requires more work on the part of the user, but gives him much more flexibility. FORMAC, on the other hand, provides a fixed set of powerful capabilities, as with a compiler; this requires far less work on the part of the user but he is then constrained by the capabilities that have been given to him and what he can build up from them.'

I feel that a mean should be found between the two. It is irritating to define the distributive law, and to apply it possibly many times throughout a program. (Notice that the example given (Page III-12) defines only one form of it. Similar transformations would have to be defined for right-handed distribution, for multiplying out the sum of three variables, and for any other instance the user required.) On the other hand, FORMAC does an automatic simplification which may not be wanted. The normal algebraic operations should be available, but should not be applied unless specifically requested.

3.1.4 Alpak and Altran.

Alpak [8] was a system designed at Bell Telephone Laboratories to do algebraic manipulations on polynomials. The language looked almost like an assembly language, and so was not easy to follow. For example,

A, B, C, and D are rational functions whose format is given by FMT. These values are presented on cards. The program to

compute the function F=(A*B/C)+D is given below. The actual program is printed in block capitals; the lower case messages on the right hand side of the page are comments.

```
RFNBEG   10000    begin(reserve 10000 words of
                  storage for data and working
                  space)
RFNRDF   FMT      read polynomial format statement
                  FMT from cards
RFNRDD   A,FMT    read polynomial A from cards
RFNRDD   B,FMT    read polynomial B from cards
RFNRDD   C,FMT    read polynomial C from cards
RFNRDD   D,FMT    read polynomial D from cards
RFNMPY   F,A,B    replace F by A*B
RFNDIV   F,F,C    replace F by F/C (C must not be
                  zero)
RFNADD   F,F,D    replace F by F+D
RFNPRT   F        print F
TRA      ENDJOB   go to ENDJOB
```

Polynomials are represented in a canonical form as ordered lists of terms. Rational functions are ordered pairs of polynomials. The functions available in ALPAK include

1) A function to find the GCD of two polynomials.

2) A function to differentiate a polynomial.

3) A function to expand a polynomial.

4) A function to get the factor of a given variable in a polynomial.

Altran [5], pages 502 - 506, [9], was an extension of this, combined with Fortran to give a readable language.

Example

```
POLYNOMIAL A1,A2,A3..
A1= (R0+R1)**2
A2 = (R0+R2)**2
..............
```

'**' and '=' are used symbolically. Not all the power of Alpak is included in Altran. Amongst the facilities omitted were means of working with truncated power series, and with systems of linear equations. The input and output of Altran was somewhat limited. For instance, the polynomial

```
xy**2 +2xyz +xz**2 + y**2 + z**2
```

was input as

```
1    1 2 0
2    1 1 1
1    1 0 2
1    0 2 0
1    0 0 2
0
```

and the output was of much the same form

```
G = numerator          x y z
                1      0 1 0
               -1      0 0 1
denominator            x y z
                1      0 1 0
                1      0 0 1
```

i.e.

$$G = (x-z)(x+z)$$

This notation is said to be easier for the user to both read and write if there are more than 100 terms.

3.1.5 An extension to PL/1.

This is a proposal for the addition of another attribute to a PL/1 variable (see [2], pages 116 - 132). The default case is NUMERIC, and FORMAL and ATOMIC are added. All variables in standard PL/1 are NUMERIC. An ATOMIC variable has its own name as its value, while a FORMAL variable is given an algebraic expression as its value. Substitution of known values takes place automatically. For example,

if A=B+1

then C=A+D

gives C the value B+1+D

where B and D are ATOMIC.

Rational arithmetic is introduced, an example of the notation being 1+2/3R for the fraction $1\frac{2}{3}$. Algebraic functions for differentation, expansion, substitution, etc. are introduced. The substitute commands suggested are similar to those of FORMAC. Four factorising functions are suggested:

1) COEFF which finds the coefficient of the expression x**i (i numeric) in a formula.

2) HIGHPOW and LOWPOW which give the highest and lowest powers of a formula.

3) GCF which, if the formula is of the form a1 + a2 + ... an, gives the highest common factor of the ai's.

The proposals describe facilities similar to those of FORMAC. However, since the new type is basic to the language, the interchange between formal and numeric variables is much freer, and therefore more desirable.

These ideas seem to be those required in our initial discussion, with the exception of pattern matching. It is worth bearing in mind that since PL/1 is already a large language, it may not be possible (or desirable) to add these ideas without abandoning some other facilities.

3.1.6 MATHLAB

Two versions of MATHLAB ([7], pages 413-422) were developed simultaneously. One was at MITRE corporation on an IBM 7030, and and the other at M.I.T using the time sharing system of PROJECT MAC. It was an interactive system, and was somewhat 'chatty'; statements like 'Thanks for variable d' were printed out in response to users' commands. Equations and expressions could be assigned to names by statements such as

```
D = 1/2*a*t**2
E == a = b*c+2
```

D is said to be a variable, and E is an equation. The algebraic commands were

please simplify(x,y)

Simplify x and put the result in y.

substitute((v1,v2,...vn) x y)

Substitute the values of the variables v1, v2,... vn in x and put the result in y.

To combine the values of variables, a rather cumbersome notation was used.

```
add((p1,p2,...pn) name)
```

add the variables p1, p2,... pn and put the result in name.

Similar instructions were available for subtraction, multiplication, division and exponentation. There were no control statements available; the interpreter worked in desk calculator mode. However, some of the algebraic commands were quite powerful.

E.g.

```
learn derivative
     arctan,
     x,
     1/(1+x**2)
```

This instructs the computer that the derivative of arctan(x) with respect to x is 1/(1+x**2)

```
integrate(v x w)
```

puts the indefinite integral

$$\int v.dx$$

into x.

```
solve(e x)
```

Solves the equation e for x. e must be linear or quadratic in x. The equation will be rearranged

first. For example, solve(e,x) where e is

1/(x**2-1) = 1/(x-1)

gives x=0, and does not find the spurious solution x=1, which satisfies this unsimplified form.

3.2 Languages Available Today

3.2.1 Altran

Altran has been constantly revised by Bell Telephones, in the last few years. From a Polynomial Manipulator, it has developed into a Rational Function Manipulator. The input/output has been altered to read and write expressions in a more mathematical form. W.S.Brown has done significant theoretical work on Polynomial Manipulation. More will be said about this in Chapter 14. Altran also has routines, and permits recursion. However it is still run in batch mode, a pre-processor being used to translate an Altran program into Fortran.

3.2.2 Mathlab

This has also been developed in the last few years. MATHLAB 68 [10] is significantly different from the original MATHLAB. First, the output is given in two-dimensional form, an appreciable improvement for long expressions. For example, the derivative of

arctan(x+a)/(1-x*b))

is printed as

```
 b(x+a)          1
--------   +  -------
       2
(1-x*b)       (1-x*b)
---------------------
                    2
               (x+a)
      1    +   ------
              (1-x*b)
```

As well as formally assigning values to names, equations and expressions are stored and can be retrieved. The latest expression to be read in is held in the workspace (ws).

A single quote mark in front of a name or function means evaluate. Thus, if y has the value x+1, then

```
z = y
```

stores y in z, but

```
z = 'y
```

stores x+1 in z.

Similarly, Deriv(x**2,x) means

$$\frac{dx^2}{dx}$$

But 'deriv(x**2,x) means 2*x.

The representation of expressions is much more natural than in the original MATHLAB. For example,

```
D: x*(y+z)
```

Stores the expression x*(y+z) as D. If an expression is not given a name, it is stored in ws.

A description of the representation of the Rational Functions of MATHLAB is given in [2], pages 86 - 97. The language is written in LISP, so the representation of an

expression is a list. A general expression is regarded as a polynomial of one variable, whose coefficients are also polynomials. For example,

x**2 + a*x*y + 7

is a polynomial in x. Its coefficients are polynomials in y. The coefficients of these are polynomials in a, whose coefficients are integers. Hence the representation of this expression is

(((1)) ((1 0) NIL) ((7)))

This representation is chosen in order to allow the polynomial factorisation algorithm to work efficiently.

Functions are available which do the following:

1) Finds the greatest common divisor of two polynomials.

2) Find the sum, product, etc. of two rational functions.

3) NEWTON. This function uses Newton's interpolation formula. Given k+1 polynomials with integral coefficients in n-1 variables, and k+1 integers, this yields a unique polynomial in n variables, such that, if

$$P_i(x_1, x_2, \ldots x_{n-1})$$

is a member of the given set of polynomials, and Q is the result, then

$$Q(x_1, \ldots x_{n-1}, n_i) = P_i$$

4) FACTOR. This factorises a polynomial with integer

coefficients into a number of polynomials irreducible over the integers.

5) BPROG. Given two relative prime polynomials, P and Q, this returns two polynomials A and B such that

$$A.P + B.Q = 1$$

6) APROG. This gives factors (with respect to the main variable) of a polynomial Q, such that

$$Q = Q_1 . Q_2^2 \ldots Q_n^n$$

where the factors Qi are pairwise relative prime, and have simple roots.

7) CPROG. This performs a partial fraction expansion on the rational function P/Q.

These functions can be used to do the following.

1) Solve an expression for a variable, x.

2) Perform linear transformations on a rational function.

3) Integrate. The program will always find the rational part of the integral.

4) Inverse Laplace Transforms.

5) Fourier Transforms.

6) Matrices and the solutions of simultaneous linear equations.

7) The solution of linear differential equations with constant coefficients.

The limitations of MATHLAB seem to be two fold; it has no subscripted variables, and it can run in desk calculator

mode only, with very few control statements available.

3.2.3 REDUCE

This was developed by A.C.Hearn, previously of Stanford University, and now at the University of Utah [12].

Reduce allows integer and real arithmetic, the latter being of arbitary precision. The arithmetic and logical and relational operators, together with the assignment character, <-, are regarded as INFIX operators. <- has the lowest precedence. New operators may be declared, and their precedence is assumed to be lower than everything except <-. However this may be changed by a PRECEDENCE statement.

Prefix operators are functions such as COS, DF (differentiate), etc.. Certain properties of COS, SIN and LOG are known. For example,

COS(0) = 1

SIN(-X) = -SIN(X)

etc.

New prefix operators may also be declared.

Strings are allowed in Reduce, as are comments.

Expressions.

Three types of expressions are allowed

1) Numeric

2) Boolean

3) Scalar (these contain algebraic variables, e.g.

X**3 - 2*Y/(2*Z**2-DF(X,Z)).

Equations are also allowed.)

Assignment.

An expression may be assigned to a variable, and the expression is normally simplified when this is done. However flags may be set to change this. In a scalar expression, real numbers are usually converted to the form INTEGER/INTEGER.

Control Statements.

1) Conditional statements.

These have the usual form of ALGOL conditional statements.

2) FOR statements have the form

```
FOR<variable> <- <arith expr>STEP<arith expr>
<term><do cl>
where
<term> = UNTIL<arith expr> or
         WHILE<boolean expr>.
<docl> = DO<statement> or
         SUM<algebraic expr> or
         PRODUCT<algebraic expr>.
```

SUM and PRODUCT return values. For example,

```
FOR I <- 2 STEP 2 UNTIL 50 SUM I**2
```

The result is held in the work space, from where it can be retrieved. The clause may also be used in an assignment, for example,

```
X <- FOR I<-1 STEP 1 UNTIL 10 PRODUCT I
```

which gives X the value 10!.

3) The GOTO statement

This references a label as in ALGOL or IMP.

4) Compound statements

These are constructed in the same way as those of ALGOL. They are enclosed in the reserved words BEGIN and END.

5) RETURN.

This returns from a compound statement to the next higher level.

6) Declarations.

Integer, real and scalar declarations may be made. Any variable not declared is assumed to be scalar.

7) Arrays.

These are declared as in FORTRAN, i.e. they have subscripts numbered from 0 upwards. Example

```
ARRAY A(10),B(2,3,4)
```

8) Flag switches

There are two statements:

```
ON <list of flag names>
OFF <list of flag names>
```

9) Commands

There are three for file handling:

IN, OUT, SHUT.

The rest are concerned with algebraic expressions.

a) LET, e.g.

```
LET X = Y**2 +1.
```

Once this declaration has been made, the

expression Y**2+1 will replace X wherever it is used.

An extension of this is

b) FOR ALL ... LET, e.g.

```
FOR ALL X LET K(X,Y) = X**Y
```

Then whenever K(?,Y) is encountered, it is replaced by ?**Y, where ? stands for any variable.

c) SUB, e.g.

```
SUB(X = X+1, Y = 1, X**2+Y++2)
```

This substitutes x+1 for x, and 1 for y in this expression only.

d) CLEAR

Removes all assignment and substitution rules.

10) Procedures

All three types may be used as procedure names. The statement

```
RETURN M
```

will give M as the result of the procedure.

Reduce is designed to be used both interactively and in batch mode. If a program is read off a file, it may contain the instruction PAUSE, which allows the user to put in instructions from the key board. The file is re-entered by typing CONT.

Other Algebraic Commands.

1) FACTOR x,y,z

Whenever a statement is simplified, factors of x, y and z are collected.

2) REMFAC

Removes this condition.

3) ALLFAC

Searches each expression for any common factor it can find.

Reduce has additional facilities such as Matrix handling routines and calculations useful in High Energy Physics. Obviously these are very useful, but are additions to the basic system, and so are not described here. Reduce was written in Lisp 1.5.

3.2.4 CAMAL

Camal [13] was developed at Cambridge University by D.Barton. Although available on the interactive system at Cambridge, the language itself is not interactive.

Two types of data objects are allowed, index expressions (iexps) which hold integers, and variable expressions (vexps) which hold algebraic expressions. The letters I ... T are used as iexps, and the remaining letters, A ...H, U ... Z for vexps. Subscripted names may also be used.

The atoms of the system are the lower case letters a, ... k,l. Examples of assignments are

A = a ** I +b

C = 12/13

Program control is very much like an Algol type program. The instructions available are

Jumps and labels

Conditional statements

For i = p:q:r; .... repeat.

A subroutine is declared by labelling the first of a set of statements. The call is

-> label ->

and it is left when the statement 'return' is encountered. Recursion is not allowed.

Subroutines are available which handle polynomials, whose general form is

$$C a^{p} b^{q} \ldots l^{t}$$

where C is either rational or real, a, b, ... l are basic atoms, and p, q, ... t are integers.

Two division operators are available.

/ will give a power series expansion.

/: divides out any known factors.

Expressions are simplified before being assigned to a variable.

Functions used for polynomials are

LCM(vexpr)

vexpr is a sum of terms, and the least common multiple of the terms is given.

HCF(vexpr)

Here the highest common factor of the individual terms is given.

COEFF(vexpr)

vexpr must be a single term, and the coefficient

is given.

NEXT

B = NEXT(A) removes the first term of A and puts it in B. For example, if

```
A = a + b + c
B = NEXT(A)
```

results in A = b + c

B = a.

EVAL(vexpr,subs)

subs is a list of values that are to be given to the variables of vexpr. These are substituted and a numerical answer is given. For example,

```
EVAL(a+b,a=0.1,b=3.5)
```

SUB(vexpr,atom,vexpr)

substitutes the first vexpr for the atom in the second vexpr. E.g.,

```
SUB(b*c,a,a+b+c)
```

gives

```
b*c + b + c
```

EXPAND(E,a,V[0],L)

This obtains the coefficients of the atom a in the polynomial E, putting the coefficients in the array V. L gives the number of coefficients required. Thus if E is

$$\sum_{r=0}^{m} b(r)*a**r$$

then

for L<m, V(0), ... V(L) are filled with b(0)...b(L)

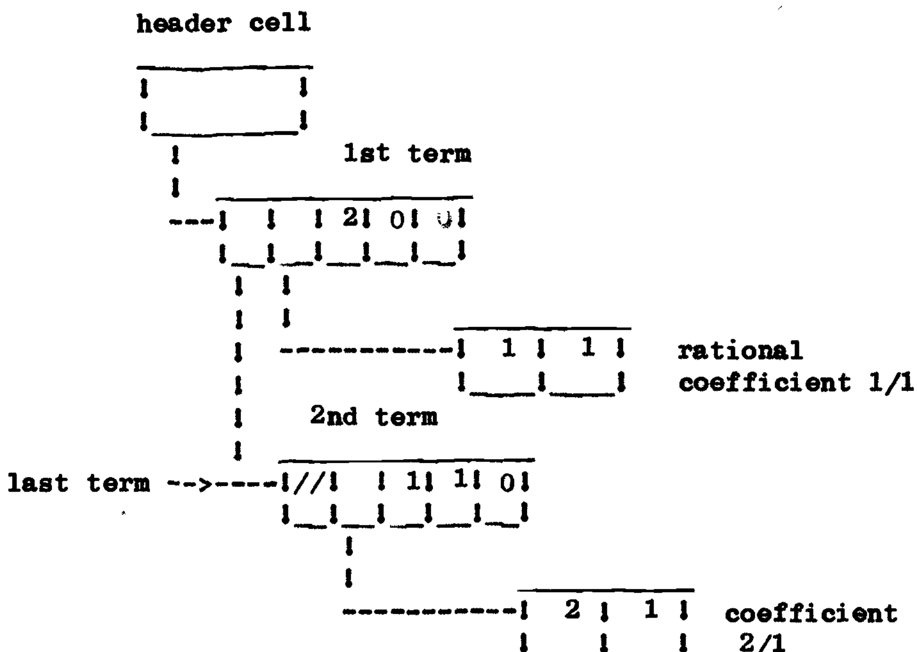
for L>=m, V(0), ... V(m) are filled with b(0)...b(m)

and V(m+1)... V(L) are filled with 0's.

CAMAL is also described in [14], [15] and [16]. The polynomials are stored in a list structure, one cell being used for each term of a polynomial. Since the atoms are known, only the exponents and coefficients are needed. Each polynomial has a header cell which contains extra information. For example, the number of references to a particular polynomial is recorded. The following is the representation of the expression

x**2 + 2*x*y.

For simplicity, it is assumed that only three atoms x, y and z may be used.



In addition to CAMAL, which is available for general use at Cambridge, there are two other structures based on polynomial handling. One is concerned with the manipulation of Fourier series, and the other with Rational Functions.

3.2.5 ALAM

ALAM, [18] and [19], (Atlas Lisp Algebraic Manipulator) was written in LISP, as its name implies, for use on the Atlas computer. It is in fact LISP with added algebraic facilities and subroutines. The operators allowed are + and * (n-ary), -(unary), and ** (exponentation, which is binary). These are used in prefix Polish lists, i.e.

```
(+ a b c)
```

represents a + b + c.

The user must program in this notation, using ordinary LISP conventions, but the output is arranged in infix form. For example,

```
COS(E) * R ** 2
```

is input as

```
(* (COS E) (** R (2 1)))
```

and output as

```
      2
COS(E)R
```

All arithmetic is rational, and numbers are stored as ordered pairs. Therefore

```
(2 1) is 2/1 = 2.
```

The exception is zero, which is written as the single atom 0.

Three functions are available for simplifying expressions. They are

ZERM which removes unwanted zeros.

EXPD which applies the distributive law

EDITMU which adds and subtracts like terms. The terms are arranged in canonical order before this is applied.
There is also a function for differentiating expressions.

The writer of ALAM is primarily interested in General Relativity, and there are several functions available that are useful in this field. However, the restriction imposed by having all programs written in LISP makes this language unlikely to appeal to the general user.

3.2.6 Scratchpad/1

This, [20], [30] pages 42-58, is a system implemented in LISP, combining facilities available in other LISP based languages, i.e. Reduce, Mathlab, Korsvold's system [1] and [21], and Martin's Mathematical Laboratory [11]. It also includes the integration program SIN by Moses [22]. It is intended for the mathematician rather than the programmer, and is used in an interactive environment.

Input is linear; and output is in two dimensional form. If used with an 1130/2250 display, Martin's Picture compiler is available.

There are five primitives of the language

INTEGERS e,g. 31, 123456789

VARIABLES x, i

FORMS   x(i), x[i] ($= x_i$)

VECTORS   {1, 2, 1, 2,}, {(1, 2), (1, 2)}

SETS   {i! x(i)=1}, {i!x(i)=1 & i>2}

There are four OPERATOR FORMS: summation (Σ), product (Π), integration ($\int$) and differentiation (∂). These have a standard linear input, for example

$$\int_{x=0}^{1} f(x)\, dx$$

is input as int[x=0:1]f(x).

An EXPRESSION is built up with primitives and operator forms, e.g.

$$x + y*(1 - u(t)^2)^{1/2}$$

Elipses(...) may be used, for example

1+2+...n,

$x^2 + x^3 + \ldots x^n$

{1,2,...n}

STATEMENTS have the syntax

VARIABLE/FORM relator EXPRESSION

where the relators are

>, >=, =, <=, <, $\in$ (belongs to)

Examples,

$f_i(x) = x ** i$,   x>0, $i \in \{1,2,\ldots\}$

These are assertions, i.e. f[i](x) has the value x**i; x is greater than zero, and i is a positive integer.

A FORM is a function or a subscripted variable. These can also be used in assertions, as for example,

i(x)=x, i(a)=a for all a.

h(x,y)=(x>y) means

h(a,b) = 1 if a>b

0 if a<=b.

Variables are assumed to range over all expressions, unless they are restricted. Hence the two statements

x>0

h(x)=x

mean that h(a)=a for all positive a.

VECTORS are ordered sets of expressions. Examples are

{u/x, 1+x, -5, 12, p(x)}

is a sequence of five elements.

{1, 2...5 }

is the first five integers.

{1, ... }

is the positive integers.

{ a1, a2, ...}

is the sequence ai, where i ranges from 1 to infinity.

{ i: j:i=j }

is the unit matrix, the size of which may be unspecified.

SETS are unordered collections of expressions, for example

S = x!x>0 & f(x)>0

Then y ∈ S asserts that

y>0 & f(y)>0

Operations on sets are not included in the current version,

but are planned for the future.

EXPRESSIONS are built up from a hierarchy of constructs.

PRIMARY

FACTOR

TERM

ALGEBRAIC EXPRESSION

CONDITIONAL EXPRESSION

PRIMARY is a variable, form, or bracketed expression, e.g.

f, fact(i), (1+u**2)

A FACTOR contains the exponential operator, e.g.

h^{i}, $u(t)^{2}$

A TERM contains multiplication and division signs, e.g.

u*v/w, t *(1-u)

An ALGEBRAIC expression contains addition and subtraction operators, e.g.

$a+b-c^{2}*d$

A CONDITIONAL EXPRESSION has an ALGEBRAIC EXPRESSION and a condition, e.g.

$x^{2}+1$ if $x>0$

An EXPRESSION is a CONDITIONAL EXPRESSION that has a ´where´ clause, e.g.

$x^{2}+f(x)-c$ if $x>0$ where $f(x)=\sin(x)-3x^{3}$

On the left hand side of a statement, the most general form allowed is a TERM. If this is used, pattern matching

rules are applied, e.g.

```
(z+phi(x)) = psi(z)
```

Then every occurrence of ?1 +phi(?2) will be replaced by psi(?1), where ?1 and ?2 are any variables. This can be used recursively, as in the example

```
cos(x+phi(y)) = mu(y)
```

Then

```
cos(phi(cos(r+phi(s)))+t)
```

is replaced by mu(mu(s)).

If only the exact term on the left hand side is to be replaced, it is enclosed in quotes, e.g.

```
''a*b'' = d
```

Then a*b+c becomes d+c,

but b*a+c

q*r+c are unaltered.

If an expression is given as a COMMAND, it is stored in the work space, and also given a number. Hence old expressions can be referred to by using this number. Expressions are evaluated before being saved. This evaluation is controlled by setting flags, with meanings as follows.

1 Simplification with no substitution.

2 Simplification with substitution.

3 Expansion under flag control.

4 Expansion in full.

5 Pattern matching from outside - in.

6 Pattern matching from inside - out.

7 Rational simplification.

8 Restructuring.

The default is 2, 3, 5, 7, and 8. This can be altered by setting evalmode, e.g.

evalmode = 1,4,8,1

means

Simplify.

Expand.

Restructure.

Resimplify.

Integration has two forms. In one the user builds integral tables, with assertions like

$$\int \frac{j_1(xt)*\sin(x)}{x} = t^{-1}*(1-(1-t^2)^{1/2}) \text{ if } t>0 \;\&\; t<1$$

$$\text{else} \quad t^{-1} \text{ if } t>1.$$

If the flag SIN is set, the SIN integrator is used.

A command may be stored by preceding it by a label of the form n.m, where n and m are integers. Stored commands may be formed into routines by statements of the form

alpha = procedure(n).

Which means that all statements beginning with n. form the procedure called alpha.

In order to add further facilities, LISP and LAP (LISP Assembly language) are available with two other facilities.

META/LISP allows users to interactively modify the existing input translators.

META/PLUS allows the syntax to be altered less formally. For example, the function for the absolute value of a variable is absval(x). The user may change the form to |x| by the statement

''|x|'' = ''absval(x)'' x expression.

3.2.7 Macsyma

This language [30], pages 59-75, was developed by W.A.Martin and R.J.Fateman. Like Scratchpad, it draws on systems already implemented at M.I.T., and is written in LISP. Martin's thesis [11] was concerned with readable two-dimensional mathematical output which can be obtained on a display or graph plotter. These facilities are incorporated in Macsyma. Mathlab has been used, and also Moses' integration program.

An expression may be stored in one of two forms:

1) as a general expression, in which case the internal representation mirrors the original expression.

2) as a rational function. This is the quotient of two polynomials, each of which is stored in a concise form. For example, 3x**2+4 is represented as

(x 2 3 0 4)

The rational function operators of Mathlab can then be used. The user can state which representation he wants.

Macsyma also has pattern matching facilities. These test for a semantic match (e.g. is the expression quadratic

in x?), rather than the syntactic match of Formula Algol.

Predicates are declared, e.g.

```
INRANGE(LOW,HI,VAR):=IF (LOW<VAR) AND (VAR<HI) THEN
TRUE ELSE FALSE
```

Then we can declare

```
DECLARE(A,INRANGE(M,N))
```

which means that the algebraic variable A has a value between M and N.

Another function DEFMATCH defines a predicate which is true only if a semantic pattern is matched. For example

```
DEFMATCH(LINEAR,A*X+B,X)
```

Then

```
LINEAR(3*Y+4,Y) is true
LINEAR(4+Y,Y) is true.
```

Two functions can be used to increase the simplification power of the language. TELLSIMP inserts a new rule before the automatic built in simplification rules, e.g.

```
TELLSIMP(COS(PI),-1)
```

will replace all instances of COS(PI) by -1. TELLSIMPAFTER puts the new rule at the end of the built in simplification rules.

3.3 Conclusions

The languages available (or described) when the AML project was started were

Van de Reit's Package

Formula Algol

Formac

Mathlab

Altran

PL/1 Extension

Of these, only Mathlab was available on-line. At that time it was a Polynomial Manipulator. The other languages were designed to be run in a batch processing environment. Formac and Formula Algol dealt with general expressions. Altran was another polynomial manipulator. It seemed that a general expression handler was preferable, as it gave the user a wider variety of problems that he could tackle. The patterns of Formula Algol were attractive because the user could extend the number of algebraic operators of the system, in a fairly straightforward manner.

Hence the requirement was for an interactive language, which could also be run without user supervision. It should handle general expressions, have the full power of a general purpose language, and should have some method of extending the range of operations offered to the user. It should also be easily readable, and produce output in a legible form. Van de Reit's package is clumsy to use, and does not satisfy these requirements. Formula Algol does not offer sufficient operations; too much is left to the user. Formac does automatic simplification, which is undesirable. Altran seemed to be in a very primitive state; although the routines behind it (Alpak) were quite powerful. It only did

Polynomial Manipulation. Mathlab worked only in desk calculator mode, and had no subscripted variables. PL/1, although a batch processor seemed to incorporated the power needed, but did not have any means of extending the operations available. No implementation of this proposal has come to my notice.

AML is an interactive language, which incorporates all the facilities of a general purpose language. In addition it is capable of running without supervision. Statements will not necessarily be lost immediately after their execution; some will be stored and executed at a later time. This method of storing statements enables facilities such as looping and routine calls, that are used in general purpose languages to be available in an interactive language as well. The ability to store statements and execute them at some later time will be known as DEFERRED EXECUTION. Pattern matching facilities are also available.

IV Some Design Features of an Algebraic Manipulation Language

In Chapter 3 we came to the conclusion that it was worthwhile exploring the possibilities of designing a language with three important features:

1) It should be capable of running interactively, with deferred exectution available.

2) It should incorporate a fair number of algebraic commands.

3) It should have useful pattern matching facilities.

General purpose languages with facilities like (1) have been written; the earliest was JOSS [23] designed at Rand Corporation in 1964. A later development is LCC [24 designed at Carnegie-Mellon University by A. J. Perlis.

The language described in this thesis is called AML (Algebraic Manipulation Language). It has four main features, each based on existing languages.

1) It is written to run in interactive mode, and has deferred execution. (JOSS and LCC)
2) It incorporates most features of a general purpose language. (IMP)
3) It contains facilities for storing algebraic formulae and commands for manipulating them. (FORMAC and FORMULA ALGOL)
4) It also contains pattern matching facilities. (FORMULA ALGOL)

The IMP programming language is currently in use at Edinburgh University. It is based on Atlas Autocode which was developed at Manchester. Its concepts and appearance are very similar to ALGOL.

4.1 Immediate and Deferred Execution.

In order to allow parts of programs to run in different modes, two types of statement are allowed. They are called LABELLED and UNLABELLED statements. If a statement is unlabelled it is obeyed immediately, thus providing an interactive, desk calculator mode. If a statement is labelled, it is stored, and may be acessed at some later time by a reference to its label. The most common way of accessing a labelled statement is to give the command

%do LABEL

which causes the statement at LABEL to be executed.

A number of statements may be grouped together by giving them a common label. In this case the command

%do LABEL

will cause the group of statements which have this label in common to be executed. It will easily be seen that a whole program can be obeyed at one time by giving all its statements a common label.

The programmer therefore has three choices. He can

1) Use only unlabelled statements, thus working in desk calculator mode.

2) Submit a program of statements all referenced by the same label, and cause them to be executed by

the command

%do LABEL.

In this case the program may be run in batch mode.

3) Write a small number of labelled statements, cause them to be obeyed, then write a few more, and proceed in this manner. This method of using the language seems the most useful during the development of a program.

4.2 Labelling AML STATEMENTS.

A statement of AML must have a unique label so that it can be accessed individually, and also a common label so that it can be accessed with a group of statements. If it is accessed in a group, there must be some way of ordering the statements within the group. It is not a good idea to use the order in which statements are presented to determine the order of execution. It is very likely that the user will accidently omit some instructions, and he must not be expected to resubmit correct instructions. Therefore it seems reasonable to require him to number all his statements to indicate the order in which they are to be obeyed.

These considerations lead to the suggestion that a label should consist of two integers, and a colon is chosen to divide them. In other words a label is of the form

INTEGER : INTEGER

Let us take as an example the label 2:1. Then the command

%do 2:1

will cause the statement at 2:1 to be executed. The command

%do 2

will cause all labels of the form 2:n to be executed, where n is an integer. The values of the n's will decide the ordering within the group.

There may be cases where a single integer is sufficient to label a statement, and so it was decided to allow the statements to take one of two forms

INTEGER

INTEGER : INTEGER

In order to avoid an inconsistency, a statement labelled with a single integer is considered to be part of the group having that integer as its common label, and will be the first of that group to be executed.

From this discussion we extend the idea of a label to consist of no more than fifteen integers, separated from each other by colons.

Examples of labels are

2:1, 3:7:1, 1, 3:1:1:1:8, 10:2:20

The integers of a label may lie in the range 1 to 255, and so they can be stored in byte integer locations.

The reason for extending a label from two to fifteen integers is to enable the user to make additions to a program without altering existing statements.

For example, suppose he wishes to insert a statement between those labelled 1:3 and 1:4. Then he could label the insertion 1:3:5. So the original program

```
1:3        i=a+1
1:4        k=j+2
```

becomes

```
1:3        i=a+1
1:3:5      j=3*i
1:4        k=j+2
```

and '%do 1' causes the statements to be executed in that order.

Now he may wish to add a number of statements between 1:3:5 and 1:4, and labels them 1:3:6, 1:3:7, ...1:3:15. An omission in this set of statements will cause him to increase again the number of integers in the label. For example, 1:3:6:1 inserts a statement between 1:3:6 and 1:3:7.

This kind of alteration is very likely to be needed when a program is being developed on-line. Allowing fifteen integers as the maximum should not impose any restriction, since beyond this number, writing the label would become unnecessarily tedious.

The program has now grown to be

```
1:3        i=a+1
1:3:5      j=3*i
1:3:6      .....
1:3:6:1    .....
1:3:7      .....
1:3:8      .....
.....      .....
```

....

1:3:15   .....

1:4   k=j+2

All these statements will be executed by the instruction

%do 1

However it will be noticed that all the statements except the last begin with 1:3. Thus the instruction

%do 1:3

would also cause all these statements except the last to be executed.

This facility can be very useful. If the user wished to test the new statements before allowing the whole program to be run, he can achieve this very simply. Instead of being labelled

1:3:6, 1:3:6:1(*), 1:3:7, 1:3:8, ...1:3:15,

the insertion could be

1:3:6:1, 1:3:6:1:1(*), 1:3:6:2, 1:3:6:3, ...1:3:6:10,

where the label marked (*) indicates the statement that was missing from the initial insertion. Then to test this group of statements, the command is

%do 1:3:6.

4.3 The Structure of AML.

The method of labelling described above imposes on AML a structure quite unlike that of IMP. Let us examine this difference in structure. IMP programs are divided into BLOCKS, which may be nested. The main program is regarded as the outermost block, and routines may be regarded as a

special kind of block. There are two points to be made about the structure of an IMP program.

1) Within a block all statements are regarded as being at the same level. Statements are executed sequentially, but a jump may be made from one statement to another, and execution continues sequentially from there.

2) A block (or routine) may only be entered from the block immediately containing it, or from a block contained in it.

AML differs from IMP principly in the fact that (2) does not hold. We can consider the unlabelled statements as defining the outermost block, and therefore analogous to the main program. However it is plain that this block does not have the properties of a main program. Moreover its behaviour is different from that of the labelled statements. These can be divided into blocks such that all those statements having a common label belong to the block whose label is that common label. Thus 1, 1:3, 1:3:2 all belong to block 1. The last two also belong to block 1:3. This means that by means of a ′%do′ instruction any statement or block can be called from any level, be executed, and control be returned to the level of the call. In other words we have a primitive kind of subroutine jump in the language.

By contrast, there is no direct ′goto′ statement. There is no obvious way of deciding what is to be done after a statement accessed by a ′goto′ statement has been

executed. Nor is it clear that such a statement is necessary. Only further experience with the system will tell whether the ommission is justified.

It seems that the use of a language structured in this way requires some adjustment in thinking on the part of the user. Programmers of IMP and FORTRAN and other such languages have been conditioned to a certain way of thinking, and this must be changed in order to use all the facilities of AML.

4.4 Features of IMP Incorporated in AML.

The features of the language which are not directly concerned with algebraic manipulation are based on statements of IMP. However because the system is designed to be used interactively several alterations have been made, so that the end result does not resemble IMP closely. In fact the method described above of labelling statements and groups of statements makes the overall structure entirely different. Similar changes were made to ALGOL in the LCC language.

The following features of IMP were thought to be necessary in AML.

1) real arithmetic

2) the cycle statement

3) conditional statements

4) routines and functions

5) recursion

1) Real arithmetic is available, although rational

arithmetic is used wherever possible, unless otherwise requested. Thus an expression containing a real quantity must result in a real expression. If an expression contains a rational, however, it may be evaluated to a real or rational expression. Rational was chosen as the default because it was felt that this mode of evaluation would be more useful in algebraic manipulations.

2) The cycle statement is changed entirely because of the structure of the language. The equivalent of the IMP cycle

```
%cycle i=1,1,10
A(i)=0
B(i)=0
%repeat
```

is

```
2:1     A(i)=0
2:2     B(i)=0
        %do 2 for i=1,1,10
```

In addition there are two other statements that cause a set of instructions to be repeated.

```
%do 2 %while CONDITION
%do 2 %until CONDITION.
```

In the first, the statements at 2 are executed repeatedly while the CONDITION is satisfied, and in the second, 2 is executed repeatedly until CONDITION is true. These statements are discussed

in more detail in Chapter 5.

3) The conditional statements look very like the corresponding IMP statements, and have four forms

```
%if CONDITION %then STATEMENT
%unless CONDITION %then STATEMENT
STATEMENT %if CONDITION
STATEMENT %unless CONDITION.
```

There is no need for %start - %finish clauses; the equivalent statement in AML has a '%do' command as the statement. For example,

```
%do 2 %unless y=0
```

4) Routines in AML must be fitted into the structure discussed earlier. The declaration of a routine is of the form

```
%routine NAME FPP %at LABEL
```

and the statements whose common label is LABEL define the body of the routine. The parameter list, FPP, is discussed in Chapter 5. There is no reason why a statement of LABEL should not also be used outside the routine. However a statement containing '%return' or '%result' would be faulted if it were not accessed through a routine call.

5) Recursion. Routines and functions may be used recursively.

4.5 Variables and Types.

The most important difference between AML and IMP is the lack of declarations in the former. It was found that

the ´type´ of a variable could be determined from its context, and so declarations became redundant. In some languages the first assignment to a variable determines its type. This is not true of AML variables; a variable may contain an integer at one time, then be assigned a real, rational or algebraic expression. It is the type of the contents that is of interest and this can be determined at the time the expression is evaluated. Variables that have nothing assigned to them are assumed to be atomic, i.e. they have their name as their value. Array declarations must be made, but only to define their bounds. There is no restriction on the type of the value of an array element. The first element could contain an integer, the second a real, and so forth. An example of an array declaration is

```
%array a(1:10)
```

Routine parameters include the five types %value, %name, %array, %routine and %fn. %Value corresponds to %real and %integer parameters in IMP, %name to %integername and %realname, and %array to %integerarrayname and %realarrayname. Similarly functions need not return a specific type. A typical function declaration might be

```
%function alpha(%value v1,%name n1) %at 5
```

All variables take values throughout the whole program, i.e. they are global. However in a routine they may be redeclared by a declaration of the form

```
%local L1, L2, ...
```

As with ´%return´ and ´%result´, this statement is only

recognised if it is accessed through a routine call. When a variable is declared to be '%local', its old value is saved, and it is free to be treated as an algebraic constant, or to have a new value assigned to it. When control leaves the routine in which it was made '%local', its current status is lost, and the value it had before the '%local' declaration was obeyed is restored. An example is given in Chapter 5.

4.6 Algebraic Expressions.

In describing the modifications made to IMP we have introduced some of the ideas concerned with the third main category of features - those concerned with the handling of algebraic expressions.

If the expression

a = b + c

is presented to the machine, then b and c, having no values assigned to them, are regarded as atomic. So a is given the expression b + c as a value.

Algebraic expressions are simplified automatically to a very limited degree. This is mainly a matter of expediency. When the system receives an expression, it has no way of knowing whether it has an algebraic expression (i.e. one that contains some atomic variables) or whether it is purely numeric. In order that one procedure may be applied to both types of expression, the following rule is observed: If two numbers occur one after the other in such a way that the precedence of the operators on either side and between them would allow them to combine, then the operator between them

is carried out. Hence if all the operands of an expression are numeric, a single number will be obtained. Otherwise a modified algebraic expression is the value of the variable.

Examples

```
6*2+3     evaluates to 15
6*2+a     evaluates to 12+a
6*a+3     is unaltered
a*2+3     is unaltered.
```

It is also permissible to label an algebraic formula thus making it a labelled statement. For example,

```
3         a+b*c.
```

This expression will not be evaluated until some reference is made to label 3. If the wrong kind of reference is made, e.g. '%do 3', the statement is faulted since 3 does not label a statement that can be executed. This method of labelling formulae is useful for handling equations that are not to be considered as assignment statements.

Example,

```
4         a+y = b+x**2-3
```

The two kinds of algebraic formulae are handled in different ways, thus providing the user with alternative methods of storing expressions. The algebraic commands of the language include:

```
%simplify
%distrib
%diff
%expand
```

In addition to these, there are several routines and functions. A full list of the commands is given in Chapter 5. Routines and functions are given in Appendix D. The argument of a command may be either a variable which has an algebraic expression as its value, or a label which labels an algebraic formula. In the first, after the command has been obeyed, the result replaces the original expression as the value of the variable. In the latter case the result is printed out at the keyboard, and the original statement is unaltered. It is possible to save the result by adding the phrase ´%into LABEL´ after the argument. This saves the result in the new label.

Example,

```
        5       a+3*a
                %simplify 5 %into 6
```

then the result is

```
        6       4*a
```

4.7 Active and Inactive Statements.

In allowing an algebraic formula to be stored as a labelled statement, AML breaks away from the usual meaning of a statement. In IMP, ALGOL and FORTRAN every statement of the language is an instruction and therefore can be obeyed. But in AML there are two kinds of statements - an instruction, which will be called an ACTIVE STATEMENT, and an algebraic formula. The algebraic formula will be said to be an INACTIVE STATEMENT.

Given one kind of inactive statement, there is no reason why other INACTIVE STATEMENTs should not be allowed. In fact any string of characters may be stored, and its validity is not questioned until it is accessed.

So far we have considered one way in which an inactive statement can be accessed: it can be used as the argument of an algebraic command. The next step is to explore other useful ways of accessing inactive statements.

When an algebraic expression has been stored as a labelled statement, it is quite possible that this expression will be required in the evaluation of another expression. As well as evaluating the statements

```
        i = a+3
        j = b+i*c
```

It should be possible to evaluate statements such as

```
1:3     a+3
        j = b+1:3*c
```

Provided that the label contains a ':', this is unambiguous, and so will be allowed. However

```
3       a+1
        j = b+3*2
```

will result in j being assigned the value b+6, since the 3 is taken to be an ordinary constant.

So far two ways of accessing inactive statements have been discussed. In one case the inactive statement must be an algebraic formula, and in the other an algebraic expression. Other inactive statements and ways of accessing

them will be introduced when they are found to be useful.

4.8 Label Expressions.

Suppose that a user wishes to execute the blocks 1:1, 2:1, and 3:1. He may of course type the three instructions

```
%do 1:1
%do 2:1
%do 3:1.
```

However the idea of causing the same block to be executed several times has already been introduced. It uses a statement such as

```
%do 1:1 %for i=1,1,3
```

Now the idea is extended to allow different blocks to be executed by writing

```
%do i:1 %for i=1,1,3.
```

Here one of the integers of a label is replaced by a variable. The more general rule would allow it to be replaced by an expression. Of course that expression must evaluate to an integer at the time of execution.

A label which may have integers replaced by expressions is called a LABEL EXPRESSION. Since %do is allowed to take a label expression as its argument, it is reasonable to allow the algebraic commands to do the same. It is also reasonable to add the cyclic %for-clause to the end of an algebraic command. Thus one may write

```
%do i:1 %for i=1,1,3
```

and

```
%simplify i:1 %into i:2 %for i=1,1,3.
```

Let us consider labels that are used in expressions, for example

```
a = b+3:1*i
```

This may be extended to allow

```
a = b+j:1*i
```

or

```
a =b+j:k*i
```

where j and k have integer values. Now let us look at the expression

```
a = b+j+1:1*i
```

This could mean

```
a = (b+j)+1:1*i
```

or

```
a = b + (j+1):1*i
```

To avoid the ambiguity, the convention is made that any expression of more than one operand that occurs in a label expression must be enclosed in brackets.

4.9 Conditions.

The conditions of AML are those of IMP with one or two additions. Some of these concern pattern matching, which will be discussed below. However one addition is the test 'i//j' which is true if i divides j, and false otherwise. I and j may be replaced by any integer valued expression.

Let us consider the instructions

```
%do 1:1 %if x=1 %and y=2
%do 2:1 %if y=1 %or z=1
%do 3:1 %if p=q %and x<0
```

```
...............
%do 4:1 %if x=1 %and y=2
%do 4:2 %if y=1 %or z=1
%do 4:3 %if p=q %and x<0
```

If conditions are allowed to be accessed as inactive statements which are accessed by replacing the condition by a label or label expression, then these statements can be written more concisely.

```
5:1    x=1 %and y=2
5:2    y=1 %or z=1
5:3    p=q %and x<0
1:2    %do i:1 %if 5:i
       %do 1:2 %for i=1,1,3
       ...............
1:3    %do 4:i %if 5:i
       %do 1:3 %for i=1,1,3
```

4.10 Patterns.

In the initial discussion on facilities required in an algebraic manipulation language, it was decided that methods for looking at the structure of an algebraic formula should be available. The pattern matching facilities of Formula Algol appear to be suitable for this kind of activity.

In the example given in Chapter 2 it was suggested that a user might want to test whether an expression is of the form

´something **2 - something-else **2´.

Clearly these facilities (described in Chapter 3) could be

used in this case. The pattern that would be used in the test would be

ANY ** 2 - ANY ** 2.

Some modification to these ideas must be made to incorporate them in AML. The symbols ´==´ and ´>>´ are already used in IMP, and so it was decided to replace them by two key words %matches and %contains. Also there is no Boolean type in IMP or AML. Hence two extra conditions are defined:

(EXPR) %matches (PATTERN)

(EXPR) %contains (PATTERN).

Examples of reserved words that may be used in patterns are

%integer, %numeric, %algebraic, and %any.

These are defined as follows

1) F %matches %integer is true if F evaluates to an integer.

2) F %matches %numeric is true if F evaluates to an integer, real or rational number, or to +infinity (INF).

3) F %matches %algebraic is true if F is an algebraic constant (i.e. is atomic).

4) F %matches %any is always true.

In addition, constants and algebraic expressions are allowed as patterns. These expressions may be mixed with key words and used in pattern expressions. So the AML test for

´something**2 - something-else**2´

is

F %matches %any**2 - %any**2 ?

In the example in Chapter 2, the user wished to discover whether a formula was of the form ´something**2 - something-else**2´, and if it was, to replace it by the product of the sum and difference of ´something´ and ´something-else´. The pattern matching facilities as described so far allow him to do the test, but not to discover what ´something´ and ´something-else´ are. In order to allow him to do this, EXTRACTORS are used. Any pattern, or operand in a pattern, may be preceded by an extractor which is of the form

(NAME) _

The condition

F %matches b_P

is defined as follows. If F is an instance of P, the condition is true and b is set pointing to F. If F is not an instance of P, the condition is false and b is unaltered.

If the extractor precedes part of a pattern rather than the whole, then b points to the part of F which matches the part of P that the extractor preceded. For example, let

F = i+1.

Then

F %matches %any+b_%integer

is true, and b points to 1 in F. Notice that b does not have the value 1; it is a pointer to the expression in F. However the part that it points to is treated as its value if it is used in an expression. So the assignment

```
a = b + 5
```

gives a the value 6.

The next example does the problem that has been discussed throughout this section.

```
1:1      F =(a+b)*(a-b)
         %do 1:1 %if F %matches a_%any**2 - b_%any**2
```

Let F=(x+1)*2-y**2. Then a points to x+1 and b points to y. In the assignment statement at 1:1, the values that a and b point to are found, substituted, and the value is stored in F which therefore has the value

```
(x+1+y)*(x+1-y)
```

Since the original expression in F has been overwritten, a and b are now undefined.

This facility can be extended further. It is quite possible that an expression may have the difference of two squares occurring in it as a sub expression, and it may be necessary to replace this without altering the rest of the expression. Two more concepts must be introduced before this can be done.

If b is an extractor pointing to part of the expression F, then that part of the expression may be replaced by another expression E by the statement

```
b <- E.
```

Suppose F = x+y+1, and b was set pointing to F by the condition

```
F %matches b_%any+1.
```

Then b points to x+y. Now if E = y*z+2, then b <- E gives F

the value

```
y*z+2+1,
```

and b is still pointing to y*z+2.

%Contains is defined in the same way as '>>' in Formula Algol. The formula F is said to contain the pattern P if there is a subexpression of F that matches P. Extractors can be used with %contains in the same way that they are used with %matches.

Example

```
                F = a+5+sin(y)
        1:1     p <- x-1
        %do 1:1 %if F %contains p_%integer+sin(%any)
```

When this has been executed, F has the value

```
a + x - 1 + sin(y).
```

The test for the difference of two squares, mentioned earlier, could be written

```
        1:1     c <- (a+b)*(a-b)
        %do 1:1 %if F %contains c_(a_%any**2-b_%any**2)
```

This can be generalised even further to find cases when the two numbers are not together.

```
        1:1:1   c <- (a+b)*(a-b)
        1:1:2   d <- 0
                %do 1:1 %if F %contains:
                c_(a_%any**2)+%any-d_(b_%any**2)
```

If F = p + x**2 + q*r - y**2 + 1,

then the result obtained is

```
F = p+(x+y)*(x-y) + q*r - 0 + 1.
```

A and b are no longer defined, but c and d are.

Patterns may also be stored as inactive statements, since some may be quite long.

Example

```
1:1       p_%any+sin(q_%integer)+r_%real+%algebraic
%do 1:2 %if x+y-z+sin(2) - 0.5 + a %matches 1:1
```

V A Full Description of AML

This Chapter describes the facilities available to the user of AML. Some of the features mentioned here have already been discussed in Chapter 4. However it was felt necessary to repeat them here in order to present a complete picture. This Chapter is also intended to be used in its own right as a Users´ Guide.

5.1 The Structure Of A Program

The labels which prefix stored statements each consist of one or more integers in the range 1 to 255, inclusive, which are separated from each other by colons, and from the statement they label by a space.

Examples

```
1:3       x = 1
2:7:53    y = 9
5         z = 26
```

In some cases it will be necessary to access only one statement. More generally, however, a group of statements will be required together. Statements which may be accessed together are said to form a BLOCK which is defined as follows. If a number of statements have the most significant part of their labels (i.e. the left-most part) in common, then they belong to the block whose label is given by the common part of those statements´ labels.

Thus, for example, statements 1:3, 1:3:2, 1:3:7:5 all belong to block 1:3. They also belong to block 1, together with statements 1:2, 1:1:5 etc. Hence there may be up to 255 primary blocks, each of which may have any number of subsidiary blocks nested in it.

This organisation provides a facility for executing small groups of statements when a tight control is necessary, and large parts of a program when a sucessful run is expected. It is possible to cause the single statement at the head of a block to be executed, and to suppress the execution of any other statements in that block by appending ´:0´ to the label in the execution instruction.

The CURRENT BLOCK is defined as the block whose label is given by removing the last colon and integer from the last label presented to the machine. For example, if the last statement was labelled 1:3:5, then the current block is 1:3.

If the next statement to be typed in is to be labelled, and in the current block, then an ABBREVIATED LABEL may be used. This has the same form as a full label, but is prefixed by a colon.

E.g. :3:2, :5.

The exact position of a statement with an abbreviated label is given by prefixing its label by the label of the current block.

Example

1:3:2 y = 1

:5 z = 12

After the first statement has been entered, the current block is labelled 1:3. Hence the full label of the second statement is 1:3:5.

After an unlabelled statement the current block is undefined, and so a full label must be used. Abbreviated labels should be used whenever possible, as the process for inserting the statement deals only with the current block. If a full label is used, the whole storage tree must be examined.

5.2 The Program

A program consists of any number of labelled and unlabelled statements which may be entered in any order. Execution will occur after an unlabelled statement is typed in, and when that instruction has been obeyed, the statement is lost and control returns to the user.

If an instruction causes a block of labelled statements to be executed, the execution of the block takes place in the following manner. Let us suppose that the block label is L. If there is a statement labelled L, it is executed. Then each of the blocks L:N which exist in the store are executed in the same manner, for N increasing from 1 to 255. This completes the execution of the block L.

Example

```
1:3:1    a = 1
:2:5     b = 6
:8       c = 10
1:3:5    d = 20
```

Let L = 1:3, i.e. we are executing the block 1:3. There is no statement 1:3, and so we must examine the blocks L:N. For N = 1, the block 1:3:1 consists only of the statement of the same label, so it is executed. Now we execute the block 1:3:2. This consists of the statements 1:3:2:5, and 1:3:2:8 which are executed in that order. Finally we execute the block 1:3:5, which again consists of one statement.

If a fault occurs as the result of executing a statement, the execution ceases, a diagnostic message is printed out, and control returns directly to the user. If a block of statements is being executed, the user can assume that all statements which should be executed before the faulty statement have been obeyed.

5.3 The Command ´%do´ And Condition Loops

The commands of the language will be discussed later. However it was felt necessary to introduce one command, ´%do´, at this point. We have discussed how a block of statements is executed without explaining how the user effects such an execution. This is done by a statement of the form

%do LABEL EXPRESSION LIST

A LABEL EXPRESSION LIST consists of a number of LABEL EXPRESSIONs, each of which may take one of the forms

EXPR1 : EXPR2 : ... EXPRn

or EXPR.

Thus a label expression has the same form as a label, except that the integers may be replaced by expressions. On execution of the statement, these expressions must evaluate to integers in the range 0 to 255, thus giving the name of a label. All the labels required are determined first and then the execution of the blocks defined by them begins. For each label, the interpreter discovers whether there is a block with that label (the program is faulted if there is not), and then executes it.

Abbreviated labels were described in 5.1. Abbreviated label expressions of the form

: EXPR1 : ... EXPRn

are defined similarly.

The first label of a LABEL EXPRESSION LIST must define a full label. Subsequent members may be abbreviated. In this context the CURRENT BLOCK is defined as the smallest block that contains the block being considered. An abbreviated label expression is prefixed by the label of the current block to give its exact position in the storage tree.

Consider for example

```
%do 1:3,:2:1,:5
```

While we are executing 1:3 the current block is labelled 1. Hence the next block we require is 1:2:1. The current block

then becomes 1:2, and so the third block to be executed is 1:2:5. As with abbreviated labels, abbreviated label expressions save time in searching the storage tree.

The statement described above may have a condition loop attached to it. This may have one of three forms, and so the statements can be of the form

%do LABEL EXPR LIST %until CONDITION (A)

%do LABEL EXPR LIST %while CONDITION (B)

%do LABEL EXPR LIST %for NAME=EXPR1, EXPR2, EXPR3 (C)

For (A) after the blocks of the LABEL EXPRESSION LIST have been executed, CONDITION is tested. (A full description of the CONDITIONS in AML will be found in 5.9). If the CONDITION is satisfied, then the whole statement has been executed. Otherwise the blocks of the LABEL EXPRESSION LIST are re-executed, the condition is re-tested, and this process continues until the condition is satisfied. For (B), the condition is tested first. No action is taken if it is not satisfied. Otherwise the blocks are executed. This process is repeated until the condition is not satisfied.

The use of (C) is similar, but this form enables us to use a counting mechanism in one statement. Before the blocks are executed, the three expressions are evaluated. They must have integer values which we shall call p, q, and r. NAME is given the value p, and the blocks are executed. If q is positive and the value in NAME is greater than or equal to r, then the execution is finished. Similarly, if q is negative and NAME is less than or equal to r, the execution

stops. Otherwise the value of q is added to NAME and the blocks are re-executed.

5.4 Assigning Values To Variables

There are no variable declarations, such as those of IMP, in AML. A name may appear in an expression, and will stand for itself as an algebraic constant, unless an assignment has already been made to it. An assignment statement is of the form

NAME = EXPR.

The value of the expression is worked out, space is reserved for NAME, and the value of the expression is stored there. Thus whenever an expression is being evaluated a check is made for each name occurring in the expression to see if it has a value. If it has, that value is substituted for the name in the evaluation.

The value being assigned to a variable may reduce to an integer, rational or real constant, or it may be an algebraic expression. Rational arithmetic is used wherever possible, unless a real value is requested by prefixing the assignment statement by the key-word ´%real´. During the evaluation if two numerical values are found next to each other, they are evaluated, provided the precedence of operators allows this. This is done because only one process is used for evaluating all expressions, and there is no way of determining whether or not an expression contains an algebraic constant prior to evaluation. No other

simplification of algebraic expressions is done by the evaluation routine. If it were, the simplified result might not show the steps of simplification in enough detail, especially if the result obtained was found to be incorrect. A seperate command is available if simplification is required.

Examples

6*3*a evaluates to 18*a

but 6*a*3 is not altered

and 6+3*a is not altered.

Let us consider some examples of assigning algebraic expressions to variables.

1) x = a + b

assigns the value ´a + b´ to x.

2) y = x + c

takes the value of x, substitutes it in the expression, and so assigns the value ´a + b + c´ to y.

However

3) a = d/2

4) p = x + z

still takes the expression ´a + b´ as the value of x. This is because the value of a variable is copied straight into the expression being evaluated, without checking to see if any of its names now have values assigned to them. If this were not so, attempting to evaluate certain

expressions could result in an infinite loop.

Consider

5) a = a + z

6) y = a + b

In (5), since the expression is evaluated before the assignment is made, a is an algebraic constant. However in (6), if a further access were made to the names occurring in the value of 'a', an infinite loop would result. As it is, y takes the value 'a + z + b'.

Returning to examples (3) and (4), the name 'a' in p could be changed by a substitution command (see 5.12) or by pattern matching (see 5.14).

5.5 Constants

As mentioned in 5.3, there are three types of numerical values available in AML. These are integer, rational and real.

Integers must be in the range -2**32 to (2**32)-1. Rationals are input in the form

INTEGER/INTEGER,

but are given a different representation inside the machine. The numerator is in the range -2**32 to (2**32)-1, and the denominator in the range 2 to (2**32) - 1. All rationals are reduced to their lowest terms before being stored. Reals can be of fixed or floating point form. The floating point form is

REAL @ INTEGER

where REAL is a fixed point real number, and the value of this is given by

REAL * 10 ** INTEGER.

5.6 Arrays

Arrays in AML may have any number of dimensions. Before an array can be used, it must be declared by a statement of the form

```
%array NAME (EXPR1 : EXPR2)
```

This declares an array, whose name is given by NAME, which is single dimensioned, and whose subscripts range from EXPR1 to EXPR2. NAME may be replaced by a list of names, if several arrays have the same bounds. A two dimensional array is declared by a statement of the form

```
%array NAME (EXPR1 : EXPR2, EXPR3 : EXPR4)
```

and similarly, for arrays of more than two dimensions, other pairs of expressions are added to the list.

The declaration may be generalised further by listing several descriptions of the form described above, in one statement.

Example

```
%array A(1:10),B,C(1:20,4:8,-4:0)
```

A rigorous definition of all statements of the language is given by the Syntax in Appendix A.

Each member of the subscript list, i.e. each

'EXPR : EXPR'

is known as a BOUND PAIR. When the declaration is executed, the two expressions of each BOUND PAIR are evaluated. Both must give integer values, and the first value must not be greater than the second.

A particular member of the array A is accessed by a phrase of the form

A(E1, E2, ...En).

When this phrase is executed, all the Ei´s must evaluate to integer values, n must be the number of dimensions of the array, and each Ei must fall in the range specified by the corresponding BOUND PAIR. Array elements may be used wherever a variable name may be used. Thus values may be assigned to them, or be retrieved from them during the evaluation of an expression. They may also be used as algebraic constants.

5.6 Routines And Functions

A routine or function is declared by a statement of the form

R/F NAME %at LB

where R/F = ´%routine´ or ´%function´

and LB is a label expression.

Hence the declaration specifies that NAME shall be the name of a routine or function and that the description of it will be found at the block given by LB. All statements of the block LB therefore belong to the description of NAME.

A routine consists of a set of instructions which will be obeyed whenever the routine is called. All the statements of the block will be executed in order, unless the statement

%return

is encountered. In this case control returns immediately to the place from which the call was made, and the statement after the call is executed next. ('%return' may be used conditionally, as described in 5.8). At the end of the block, if no '%return' has been encountered, control likewise passes to the statement after the call.

The routine is called by a statement consisting of just its name, and this statement may also be used conditionally.

A function also consists of a block of instructions, but on its return to the place from which it was called, it must give a value. This value is given by a statement of the form

%result = EXPR.

All function descriptions must contain at least one statement of this form, and this too may be used conditionally.

The name of a function may be used in any expression, as with the name of a variable. However, instead of retrieving a value immediately, the statements of the block holding the description are executed until a 'result' statement is reached. Then the value of the expression given there replaces the function name in the evaluation of the original expression. Obviously no name will be recognised as

a routine or function name until the declaration has been executed. Also the program will be faulted if a call is made and the description has not been stored in the block specified.

5.6.1 Parameters

Routines and functions are used when the same piece of code is required several times in the course of executing a program. However this facility can be made much more powerful by the use of PARAMETERS. A statement of the form

%routine NAME (%value NAMES) %at LB

declares a routine whose name is given by NAME, and which has a number of VALUE TYPE PARAMETERS listed by NAMES. The call of the routine would be of the form

NAME (EXPR LIST)

where the number of expressions in EXPR LIST is the same as the number of names in NAMES. NAMES is said to be a list of FORMAL PARAMETERS while the expressions listed in the call of the routine are ACTUAL PARAMETERS.

The description of the routine will contain statements referring to the names given in the formal parameter list. When the routine is called, the expressions of the actual parameter list are assigned to the names of the formal parameter list, in order, and these are the values that are used in executing the routine. On exit from the routine these values are lost, and the names used in the formal parameter list have the status that they had before the routine call was made.

Example 1

```
                %routine add(%value i,j,k) %at 1:1
      1:1       n = 0
      1:1:1     %do 1:1:4 %for m = i,1,j
      :4        n = n + A(m)
      :2        A(k) = n
      :3        %return
```

This is a routine which adds the values of A(i) to A(j) inclusive, and puts the total in A(k). A is assumed to have been declared as an array. Thus

```
          add(1,3,5)
```

will put the value of A(1) + A(2) + A(3) into A(5).

```
          i = 4
          add(i,i+5,20)
```

will put the value of A(4) + ... A(9) into A(20).

Example 2

Functions as well as routines may use parameters. Thus we could modify example 1 to be a function.

```
                %function add(%value i,j) %at 1:2
      1:2       n = 0
      1:2:1     %do 1:2:3 %for m = i,1,j
      :3        n = n + A(m)
      :2        %result = n
```

This time the total A(i) + ... A(j) is returned as the result of a function. The call of the function would be in an expression, as for example

```
          z = add(1,3) + 2
```

which puts the value A(1) + A(2) + A(3) + 2 in z.

There is another type of parameter, known as a NAME TYPE PARAMETER which acts rather differently. A typical routine declaration involving name type parameters is

%routine NAME (%name NAMES)

where NAMES is a list of names. The call of this routine would be

NAME(NAMELIST).

NAMELIST is also a list of names, which must be the same length as NAMES. This time a link is set up between each name of the formal parameter list and its corresponding member of the actual parameter list. If any member of the actual parameter list has not been assigned a value, space is set aside for it, as if an assignment were about to be made.

In the execution of the routine description, whenever the name of a formal parameter is used, the name of the actual parameter is accessed via the link, and a value is either read from or written to this variable. Both name and value type parameters may be listed in a routine description.

Example 3

We could modify Example 1 in another way, so that instead of insisting on the total being stored in a member of A, we can specify the variable to which it will be assigned.

```
%routine add(%value i,j,%name k) %at 1:3
1:3      n = 0
:3:1     %do 1:3:4 %for m = i,1,j
:4       n = n + A(m)
:2       k = n
:3       %return
```

The statement

```
add(1,3,x)
```

puts the total A(1) + A(2) + A(3) in the variable x. Obviously any alteration to the formal parameter inside the description will alter the actual parameter and this effect is not lost when we return from the routine.

Arrays, functions, routines, and labels may also be used as parameters. The first three are similar to %name type parameters, and the last is more like %value type parameters. The declarations for the first three are

```
%routine NAME(%array NAMELIST) %at LABEL EXPR
%routine NAME(%fn NAMELIST) %at LABEL EXPR
%routine NAME(%routine NAMELIST) %at LABEL EXPR.
```

Of course these types of parameters may also be used in functions. The call of the routine in all these cases would be

```
NAME(NAMELIST1)
```

where NAMELIST1 is also a list of names, of the same length as NAMELIST. Furthermore these names must be the names of declared arrays, functions, or routines, according to the routine declaration. In all cases, whenever a formal

parameter appears in the block specified by LABEL EXPR, the name is replaced by the actual parameter given in the call. The subscript list following the name must correspond to the subscript list expected for the actual parameter. Hence in the case of arrays, the number of subscripts must be the number of dimensions of the actual parameter. Similarly for routines and functions the subscript list must correspond to the parameter list expected for the actual parameter, in length and in type of parameter.

Example 4

Example 1 restrains us to adding members of the array A. This can be extended to do addition for any array of one dimension.

```
                %routine add(%array A,%value i,j,k) %at 1:4
     1:4        n = 0
     :4:1       %do 1:4:4 %for m = i, 1, j
     :2         A(k) = n
     :3         %return
     :4         n = n + A(m)
```

Then add(B, 1, 5, 50)

will put the value B(1) + ... + B(5) into B(50).

Example 5

The example for a function parameter will have to be different from the basic example that has served our purpose so far. Let us suppose that we wish to fill some array with the results of some function. The routine to do this could be

```
%routine fill(%array B, %fn f, %value i,j) %at 1:5
1:5      %do 1:5:2 %for m = i, 1, j
:5:1     %return
:2       B(m) = f(m)
```

Then fill(A, fact, 1, 10)

could be used to fill A(1) to A(10) with the values 1! to 10!.

```
fill(C, log, 20, 35)
```

would fill C(20) to C(35) with the values log(20) to log(35).

Example 6

To adopt the above example to illustrate the use of a routine parameter is slightly artificial. However let us suppose that instead of functions we have a number of routines that take one value type parameter, and return a value via a name type parameter. One of these routines could be declared as

```
%routine f(%value i, %name k) %at 2:1
```

Then Example 5 could be altered in the following way.

```
%routine fill(%array B,%routine g,%value i,j):
%at 1:6
1:6      %do 1:6:2 %for m = i, 1, j
:6:1     %return
:2       g(m,k)
:2:1     B(m) = k
```

and this would be called by

```
fill(A, f, 1, 10).
```

The declaration of a routine with %label parameters is of the form

```
%routine NAME(%label NAMESLIST) %at LABEL EXPR
```

and the call for this is

```
NAME(LABEL EXPR LIST).
```

Again the lists must be of the same length, and as before any occurrence of the formal parameter in the block describing the routine is replaced by the actual parameter. This facility is useful to reference statements that are accessed by their labels.

Example 7

Instead of having a number of routines as in Example 6, we could have a number of blocks to be executed, and wish to put the value of a particular variable into an array. One such block could be

```
3:1:1       n = 1
:1:2        %do 3:2 %for m = i, 1, k
3:2         n = n * A(m)
```

Now we could declare the routine

```
%routine fill(%array B,%label L,%value i,j,:
%name k) %at 1:7
1:7         %do 1:7:2 %for k = i, 1, j
:7 1        %return
:2          %do L
:2:1        B(k) = n
```

The call would be

```
fill(C, 3:1, 1, 10, n)
```

This puts the values

A(1) into C(1)

A(1)*A(2) into C(2)

A(1)*A(2)*A(3) into C(3)

etc.

5.6.2 Local Declarations

In the first three examples, besides the parameters, we have used two other variables, m and n, whose final value is of no significance outside the routine. We can use two different names every time we describe a routine, but to save the number of names required it is preferable to restrict the range of validity of the two names themselves. This done by the statement

%local m,n

or, in its general form

%local NAMELIST.

This statement should appear as the first of the routine description. Then any values that the names of NAMELIST may have are stored away, and the names are ready to be used in any way desired. Then on exit from the routine, the values that the names have are lost, and the previous values are restored. The names that are declared ´%local´ need not necessarily be used as variables outside the routine. They could be the names of arrays, routines and functions as well. Similarly it is possible to set up arrays which will only be used inside the routine by prefixing the array declaration by ´%local´.

E.g.   %local %array A(EXPR1 : EXPR2)

Functions and routines may be used similarly.

%local function A %at 2:1.

Note that the description of the local function need not be inside the block describing the routine in which it is defined. For example

```
            %function alpha %at 1:1
   1:1      %local %function beta %at 2:1
   :1:1     %result = beta + 5
```

Here the local function ´beta´ is found at block 2:1, which is outside the block describing ´alpha´. Outside ´alpha´ the routine beta is not defined, and so the block 2:1 can be accessed by a ´%do´ command or may describe another routine, if that is required. It can, of couse be ignored and not accessed at all outside ´alpha´.

5.7 The Dynamic State Of The Program

Generally, in a language that uses a compiler, the structure of the program is determined statically before the program is run. The type and scope of a variable, the descriptions of routines and functions are all determined at compile time, and are fixed from then on. With an interpreter the situation is completely different. Such things are determined dynamically. Thus a routine is recognised as such immediately its declaration has been executed. The description of a routine or function may be altered between one call and the next provided it satisfies

the required restraints. Statements may be overwritten; submitting a statement with the same label as a statement already stored causes the old statement to be overwritten. The type of a name may be changed. Thus the declaration

```
%array A(1:10)
```

will erase any value that was assigned to A and set up the array. Similarly if we wish to reset a name to its original un-assigned state we may use a statement of the form

```
%empty NAMELIST.
```

Then all the names of the list, whether they named variables, arrays or functions, are reset to algebraic constants.

No statement takes effect until it has been executed. Unlabelled statements are executed immediately, and so cause no problem. However labelled statements must be executed explicitly by means of the command ´%do´. For example

```
1:3     %array a(1:10)
        a(5) = 7
```

will be faulted, since a is not recognised as an array. The correct version is

```
1:3     %array a(1:10)
        %do 1:3
        a(5) = 7.
```

5.8 Conditional Statements

A conditional statement can be written in the form

```
%if CONDITION %then INSTRUCTION.
```

If the CONDITION is satisfied, the INSTRUCTION is obeyed. Otherwise no action is taken, and control passes to the next statement. INSTRUCTION may be replaced by any of the statements described as an UNCONDST (unconditional statement) in the Syntax in Appendix A. The conditions will be discussed in 5.9.

Another form of conditional statement is

%unless CONDITION %then INSTRUCTION.

This time the INSTRUCTION will be obeyed only if the CONDITION is not satisfied. These two statements may be extended by the addition of an ELSE CLAUSE. E.g.

%if CONDITION %then INSTR1 %else INSTR2.

If CONDITION is satisfied INSTR1 is obeyed; if it is not, INSTR2 is obeyed. Similarly for

%unless CONDITION %then INSTR1 %else INSTR2

INSTR2 is obeyed if the condition is satisfied, and otherwise INSTR1.

The first two conditional statements may be shortened by turning them round into the forms

INSTR %if CONDITION

INSTR %unless CONDITION.

However if an else clause is required, the first form only may be used.

5.9 Conditions

In the above section we have discussed the use of conditional statements without defining what CONDITION stands for. Obviously we require some range of tests in order to direct the flow of control in the program. The simplest conditions are

EXPR1 = EXPR2 (1)

EXPR1 # EXPR2 (2)

In both these cases the two expressions are evaluated. If the results are both numeric, case(1) is satisfied if the values are the same, and case(2) if they are not. If one expression is numeric and the other is not, case(2) is satisfied. If both are algebraic, case(1) is satisfied only if the evaluated expressions are identically equal. Otherwise case(2) is satisfied. Thus for example 'a + b' would not be regarded as equal to 'b + a' in this context.

For numerically valued expressions we have the range of inequalities.

EXPR1 > EXPR2 (3)

EXPR1 >= EXPR2 (4)

EXPR1 < EXPR2 (5)

EXPR1 <= EXPR2 (6)

We may also have the double sided inequalities

EXPR1 > EXPR2 > EXPR3 (7)

EXPR1 < EXPR2 < EXPR3 (8)

and these may further be extended by replacing either or both '<' signs by '<=', and similarly for '>'.

For integer valued expressions there is a further case

EXPR1 // EXPR2 (9)

This case is satisfied if EXPR1 exactly divides EXPR2, i.e. if it is a factor of EXPR2.

Two other conditions, the pattern conditions, will be discussed in 5.14.

These nine cases, together with the pattern conditions are said to be SIMPLE CONDITIONS. Simple conditions may be joined together to form a CONDITION by means of the key-words '%and' and '%or'. Thus

SC1 %and SC2 (i)

is satisfied if both SC1 and SC2 are satisfied.

SC1 %or SC2 (ii)

is satisfied if one or both of the two simple conditions is satisfied. As many simple conditions as required may be strung together using '%and' and '%or', but the two key words may not be mixed, as this would give an ambiguous condition.

E.g. x > 0 %and y > 1 %or z < 5.

To define the meaning clearly it would be necessary to indicate which pair of conditions should be considered first.

Thus the result of

(x > 0 %and y > 1) %or z < 5

is not necessarily the same as that of

x > 0 %and (y > 1 %or z < 5).

To incorporate these more complex conditions we add a

further case to the simple conditions.

(COND) (10)

where COND is any condition, and this can be used with other simple conditions in expressions of the forms (i) and (ii).

Examples

```
%if x > 0 %and y = 1 %then %do 1
x = 0 %unless y < 0 %and (z = 2 %or z = 0)
%if z=2 %or y=1 %or x=0 %then a=3 %else a=5
```

are all examples of conditional statements.

5.10 Active And Inactive Statements

When a statement is to be executed, the syntax analyser is called to determine whether or not the statement is an ACTIVEST, as given by the syntax in Appendix A. If the statement is recognised as an ACTIVEST, it is executed. Otherwise, the diagnostic message 'SYNTAX' is printed out beneath it and control returns to the user. The faulty statement is printed, if it is labelled.

Hence we have two uses for labelled statements. They may be instructions which will at some time be executed (activated by the command '%do') or they may contain data that will be used by active statements. Certain active statements of the language may refer to a labelled statement by its label. These statements fall into two types. Those statements which perform the algebraic operations such as simplification, binomial expansion etc., may have a label or label expression as their operand. The statement given by

the operand is expected to be an algebraic expression or equation, and the operation is performed on this, the result being stored elsewhere so that the original statement is not overwritten.

In the second case an active statement which uses a labelled statement incorporates the contents of that statement into itself before being executed. In certain places where part of an active statement is liable to be long or to be changed, that part of the statement may be replaced by a label expression. Then on execution of the statement the labelled statement replaces the label, and provided the part is syntactically correct, the active statement is executed. A CONDITION is one such part that may be replaced by a label.

Example

```
1:4 x > y %and ((z - 1) * 3 > 2 * a %or a * z < 5 * y)
    %do 1:3 %if 1:4.
```

An operand in an expression may also be replaced by a label. This is necessary since the algebraic operators work on labelled statements.

Example

```
1:3   (a + b) ** 4
      %expand 1:3 %into 1:4
```

(The binomial expansion of ´(a + b) ** 4´ is to be placed in 1:4)

```
      d = 1:4 * 2.
```

5.11 Other Unconditional Statements

a) %print (TEXT)

(TEXT) is defined as any string of characters enclosed in quotes. %print causes the text (without the quotes) to be printed. The occurrence of two adjacent single quotes in the text string results in a single quote being printed.

Examples

```
    %print 'an example'
prints
    AN EXAMPLE
    %print 'fred''s example'
prints
    FRED'S EXAMPLE
    %print '''stop'' he said'
prints
    'STOP' HE SAID
    %print ''''
prints
```

b) %stop

Stops executing the current block, and returns to the main level to obtain the next statement from the user.

c) %exit

Returns from the current block to the block that called it, continuing immediately after the statement that called it.

Example

```
1:1 %do 2
:2  write(i)
2:1 %exit %if i=0
:2  i=j/i
    j=5; %do 1
```

Then for i = 0, 0 is printed, otherwise 5/i is printed.

d) %finish

Exits from AML.

5.12 Commands

It was found necessary to discuss the command '%do' at quite an early stage. The rest of the commands will be described here. They can be divided into two groups - general commands and algebraic commands.

5.12.1 General COMMANDS

1) %write LABEL EXPRESSION LIST

This command causes the blocks given by the label expressions of LABEL EXPRESSION LIST to be printed out at the terminal. Each block is printed as it is held in the storage tree, i.e. in the order in which it would be executed. Hence if the statements of a block have been presented to the machine at different times, and in no set order, it is often useful to use this command before executing a block, to ensure that the statements in the block are correct.

2) %erase LABEL EXPRESSION LIST

This command deletes all the statements of the blocks listed. The labels themselves are not removed, and so the fault 'STATEMENT MISSING' not 'LABEL NOT SET' will appear if one tries to use a statement that has been removed by '%erase'. This command is useful, not only for deleting faulty statements, but also to ensure that no stray statements have been left in a block that is about to be re-written. To erase a single statement, one should append ':0' to the label expression defining it. This will ensure that no other statement of the block is erased.

3) %label LABEL EXPRESSION LIST %as LABEL EXPRESSION LIST

Let LB1 name the first LABEL EXPRESSION LIST, and LB2 the second. Then this command copies the blocks given by LB1 into LB2 in a manner that preserves the structure of each block. The original block is unaltered in each case. Thus if the block labelled L is to be labelled as the block M, let L:N be the label of a particular statement. L, M and N all have the form of a label. Then the new label of the statement is M:N. An example will make the process more clear.

Example

```
1:3       a = 1
:3:1      b = 2
:5        c = 3
:2:1      d = 4
1:3:1:5   e = 5
```

```
%label 1:3,:3:1 %as 2:7, 1:6

2:7       a = 1
2:7:1     b = 2
2:7:1:5   e = 5
2:7:2:1   d = 4
2:7:5     c = 3
1:6       b = 2
1:6:5     e = 5
```

The first block to be labelled is 1:3, and so all the statements of the block have their first two integers replaced by 2:7. Notice that although they were not entered in order, the print-out at the terminal is ordered. Next the block 1:3:1 is labelled 1:6, and so we have the last two statements. Note that the two LABEL EXPRESSION LISTs must be the same length.

4) %read NAMELIST* %from %file EXPR

This command enables the user to read in sets of data from sources other than the terminal at which he is working. EXPR should be integer-valued, thus giving the stream number of the file. The connection between physical files and stream numbers is made outside AML. (For details, the user is referred to the HELP information of EMAS).

NAMELIST is defined as NAME LIST?, and hence identifies a list of variables and array elements. Whenever the command is executed, if the length of NAMELIST* is n, then the first n expressions in the

file are read into the locations defined. These values are then lost from the file, and the next time a '%read' command is executed for the same file, the (n+1)-th expression is taken as the first available. The expressions on the file may be any acceptable AML expressions, and are separated from each other by at least one space, or by a newline symbol. An expression is evaluated before being assigned to the appropriate location.

Example

```
    1:1      %read a,b,c(i) %from %file 3
    :1:1     ..........
             ..........
             %do 1:1 %for i = 1,1,2
```

Let the contents of file 3 be

2 5 7 p+q b+3 d

The first time 1:1 is executed, a, b, c(1) are given the values 2, 5, 7, respectively. The second time, a is given the algebraic expression 'p+q'; b+3 is evaluated to 8 before being assigned to b, and c(2) takes the algebraic constant d.

5) %eval (NAME LIST)*

It has been noted above that while all known variables are substituted in an expression during the execution of an assignment statement, no attempt is made to alter the value of an expression, if variables which were algebraic constants are later given values. For example,

```
a=b+c
d=a+g
```

results in d having the value ´b+c+g´. However, the assignment

```
g=5+x
```

does not alter d. The reasons for choosing this approach are given in the discussion on the assignment statement.

However it is possible that the substitution of its value is required for a particular variable. The command

```
%subs g %in d
```

would do this. %Eval gives a further facility, in that EVERY name occurring in its argument is replaced by its value (if it has one). The exception to this rule is the argument itself (for obvious reasons). Thus consider

```
g=y + z*x + g
y=1/2
z=a+b
x=5-x
%eval g
```

This would result in g having the value

```
1/2 + (a+b)*(5-x) + g.
```

A further example deals with arrays and functions whose values may also be substituted.

```
g = y(x) + h(i)
i = 2
%array h(1:10)
h(2) = b + c
```

```
%function y(%value a) %at 1:1

1:1 %result = a**2/3

%eval g
```

Gives g the value

```
x**2/3 + b + c
```

Example 4 of Appendix E shows a useful application of %eval.

5.13 The Algebraic Commands

5.13.1 The Into Clause

We have indicated above that algebraic commands may use labelled statements as operands. The result of applying a command to such a statement does not alter the original statement, but is stored elsewhere. If an INTO CLAUSE is used, this specifies where the result is to be placed. Thus we have a statement of the form

COMMAND LABEL EXPR LIST %into LABEL EXPR LIST.

The lists must be of the same length. All the statements of each block of the first LABEL EXPR LIST (LB1) have the command applied to them, and the results are stored in the corresponding block of the second LABEL EXPR LIST, so that the structure of the original block is preserved. (As for ´%label´.)

The statement form given above can be extended to

COMMAND FLBLIST %into LABEL EXPR LIST (2)

where FLBLIST is a list which may contain both label expressions and formulae. If a member of the list is a label expression, the block given by the label is used as an

operand, as described above. If a member is a formula, then it is used as the operand, and the result is stored in the label given by the corresponding member of the LABEL EXPRESSION LIST.

One may also apply algebraic commands to variables or array elements that contain algebraic expressions. The form of the statement is

COMMAND NAMELIST

where NAMELIST defines a list of variables and array elements. This time the result of applying the command is assigned to each variable, thus overwriting the original value. If an INTOCL is appended to a command with a NAMELIST, the result is put in the labels specified, and the contents of the variables are left unaltered.

Examples

i)
```
a = 3 * x + 2 * x
b = 5 + y + 3 + 2 * y
%simplify a,b
```

then

a now has the value 5 * x

and b has the value 8 + 3 * y.

ii)
```
%simplify c 5into 1:1
```

leaves c unaltered, and the result is in 1:1

(See below for a description of ´%simplify´).

5.13.2

Now we shall discuss the individual commands.

1) %simplify INTOCL

The INTOCL may be one of the two forms described above, i.e.

FLBLIST %into LBLIST

or NAMELIST.

This command simplifies its operand according to the following general rules.

i) All numerical calculations are carried out.

ii) Like terms with numerical coefficients are collected together.

Let S be an algebraic expression to which %simplify is to be applied. We shall discuss the results of applying the command to various examples of S, giving a general form of the result and also specific examples. In the general form a(i), b(i), c(i) will be used for algebraic variables, and p(i), q(i), r(i) for numerical variables, unless otherwise stated. If an expression involving p, q, r has these letters in upper case, it indicates that the whole expression is numeric, and so is evaluated.

a) S = a ** b(1) ** b(2) ** ... b(n)

result = a ** (b(1) * b(2) * ... b(n))

If an algebraic variable is raised to a power by a sequence such as is given above, these terms are multiplied together to give the form given by 'result'.

E.g. S = a ** b ** c

result = a ** (b * c)

The term inside the bracket is also simplified according to rules (c) and (d) given below.

E.g. S = a ** b ** 3 ** b

result = a ** (3 * b ** 2)

S = a ** b ** (c * d / b)

result = a ** (c * d)

b) S = p ** b(1) ** ... q(1) ** ...b(m) ** ... q(n)

result = (P ** (Q(1) * ... Q(n)) ** (b(1) * .... b(m))

If a numeric variable is raised to the power of the series as in (a), any numerical variables of the series are removed, and p is raised to the power obtained by multiplying them together. The remaining values are multiplied together as in (a).

E.g. S = 3 ** a ** 2 ** b

result = 9 ** (a * b)

c) S = a(1) * p(1) * ... a(m) * ... p(n)

result = (P(1) * ... P(n)) * a(1) * ... a(m)

Numerical coefficients are taken to the front of the term.

E.g. S = a * b * 3 * c * 4

result = 12 * a * b * c

S = a ** b * 3 * c * 2

result = 6 * a ** b * c

Here the a's may be algebraic variables, expressions to which rules (a) and (b) have been applied or bracketed subexpressions.

d) S = a(1) * a(2) * ... a(n)

If for some i, a(i) = b ** p(i) and if there exists

a(j) = b ** p(j) then

result = a(1) * ... b ** (P(i) * P(j)) * ... a(n)

E.g. S = b ** 3 * c * b ** 2

result = b ** 5 * c

S = b * c * b **3

result = b ** 4 * c

S = b * c / b * d

result = c * d

e) S = p(1) + a(1) + ... p(m) + ... a(n)

result = (P(1) + ... P(m)) + a(1) + ... a(n)

Numerical values are added together and put at the beginning of the expression.

E.g. S = a + 3 + c + 5

result = 8 + a + c

S = a ** 2 + 4 + c * d + 5

result = 9 + a ** 2 + c * d

The a´s are algebraic constants, expressions evaluated by rules (a) to (d) or bracketed subexpressions.

f) S = a(1) + a(2) + ... a(n)

If for some i, a(i) = p(i) * a and if there exists

a(j) = p(j) * a then

result = a(1) + ... (P(i) + P(j)) * a + ... a(n).

In applying this rule, the law of commutativity is used. Hence if

a(i) = p(i) * a(1) * a(2)

and   a(j) = p(j) * a(2) * a(1)

then the result is

.... (P(i) + P(j)) * a(1) * a(2)....

E.g. S = 3 * a + b + 2 * a

result = 5 * a + b

S = 3 * a * b + c + b * a

result = 4 * a * b + c

2) %real %simplify INTOCL

This command has the same effect as (1) except that real arithmetic is used for all rational numbers.

3) %distrib INTOCL

This command applies the distributive law to all its operands in the following manner. Let S be an operand of the command. Then if

S = (a(1) + a(2) + ... a(n))*(b(1) + ... b(m))*c

result = a(1)*b(1)*c + a(2)*b(1)*c + .. a(n)*b(1)*c

+ .... a(1)*b(2)*c + ... a(n)*b(2)*c

+ ...... a(n)*b(m)*c.

The a´s, b´s, and c are algebraic expressions not involving the operators ´+´ and ´-´. The resulting expesssion is simplified. S may be extended to contain any number of c´s, and any number of bracketed expressions of the form given by the a´s and b´s. Any ´+´ operator may be replaced by ´-´. If the operand is a sum of terms of the form described above, each one of these terms has the rule applied to it, before the

whole expression is simplified.

Examples

S = 3 * (a + b)

result = 3 * a + 3 * b

S = 3 * (a - 2 + b) * (b - a + 4)

result = 3*a*b - 6*b + 3*b ** 2 - 3*a ** 2 + 6*a

- 3*a*b + 6*a - 24 + 12*b

which simplifies to

6 * b + 3 * b ** 2 - 3 * a ** 2 + 12 * a - 24.

S = a * (b + c) + b * (a - c) - c * (b + a)

result = a*b + a*c + b*a - b*c - c*b - c*a

= 2 * a * b - 2 * b * c

The command is also applied to any sub-expressions with the same structure.

E.g. S = a * (b * (c + 2) + d)

result = a * b * c + a * b * 2 + a * d

4) %expand INTOCL

This applies the multinomial expansion to any parts of its operand that are of the form

S = (a(1) + a(2) + ... a(n)) ** p

where p is integer or rational valued.

Thus for positive integer p

result = a(1) ** p + ... C(p,r) * a(1) ** (p - r) *

ev((a(2) + ... a(n)) ** r) + ...

ev((a(2) + ... a(n)) ** p)

where ev(A) is the result of applying '%expand' to A, and

C(p,r) = p! / (r! * (p-r)!).

If p is a negative integer, or rational, we must expand a term of the form

(1 + b) ** p.

Hence the general form is put into the form

(1/(a(1)**p) * (1 + (a(2) + ... a(n))/a(1))**p (1)

The expansion in this case is infinite, and in AML, unless otherwise stated, (by adding a rider ´%to EXPR %terms´), only the first five terms are considered.

Hence putting

b = (a(2) + ... a(n))/a(1)

we obtain

(1/(a(1)**p)) * (1 + f(p,1)*b + f(p,2)*b**2 + ...

f(p,5)*b**5) (2)

as the expansion of (1), where

f(p,r) = (p * (p - 1) * ... (p - r + 1))/r!

The result of %expand is given by applying %distrib to (2).

Examples

S = (a + b) ** 3

result = a**3 + 3*a**2*b + 3*a*b**2 + b**3.

S = (a + b + c) ** 4

result = a**4 + 4*a**3*b + 4*a**3*c + 6*a**2*b**2

+ 12*a**2*b*c + 6*a**2*c**2 + 4*a*b**3

+ 12*a*b**2*c + 12*a*b*c**2 + 4*a*c**3

```
+ b**4 + 4*b**3*c + 6*b**2*c**2
+ 4*b * c**3 + c**4.
```

Multinomial expansions are treated as binomial expansions, with the 2nd to n-th terms bracketed together. Thus (a+b+c) is treated as (a+(b+c)). Hence the way in which the above example is evaluated is as follows.

```
(a + b + c)**4 = (a + (b + c))**4
```

which is

```
a**4 + 4*a**3*(b+c) + 6*a**2*(b+c)**2 +
4*a*(b + c)**3 + (b+c)**4
```

Expand is applied again to (b+c)**2, (b+c)**3, and (b+c)**4, and then the terms are expanded out.

Example

```
1:1      (1 + x) ** (-1)
         %expand 1:1 %into 1:2 %to 6 %terms
1:2      1 - x + x**2 - x**3 + x**4 - x**5 + x**6
```

In some cases one may not require the whole expansion, but only a particular term. In this case the command is of the form

```
%expand INTOCL %for EXPR %th %term
```

EXPR must be integer valued at the time of execution. Let its value be n. Then each part of the operand that is of the form S, given above, (page V-40) is replaced by the n-th term of the expansion in the result. The term is ommitted altogether if p is a positive integer less than n, since in this case there is no n-th term.

Example

```
1:1       (a + b) ** 2 + (1 + a) ** 1/2

          %expand 1:1 %into 2:1 %for 4 %th %term

2:1       1/16 * a

1:2       (a+b)**4 + (a+c)**5

          %expand 1:2 %into 2:2 %for 4%th %term

2:2       4*a*b**3 + 10*a**2*c**3

1:3       (a+b+c)**4

          %expand 1:3 %into 2:3 %for 3%rd %term

2:3       6*a**2*b**2 + 12*a**2*b*c + 6*a**2*c**2
```

In the last example (a+b+c) is treated as (a+(b+c)), and the third term of this binomial expression is found. (b+c)**2 is then evaluated.

5) %addsum INTOCL

This command searches its argument for any terms of the form

```
power(A(r), x, r, L, U)
```

where A(r) is any function of r. This is a recognised function representing the summation of

```
A(r) * x ** r
```

for values of r between L and U, including those values.

If U=INF, the summation is from L to infinity. Similarly if L=-INF, it is from -infinity to U, and

```
power(A(r), x, r, -INF, INF)
```

represents the summation from -infinity to +infinity. The term may be multiplied by any factor not involving

x or r, and if this is the case, the factor is absorbed into A(r) before any addition takes place. Thus

```
p * power(a(r), x, r, L, U)
```

is regarded as

```
power(p * a(r), x, r, L, U)
```

If there is more than one such term in the operand, and if the bounds overlap, then the terms are added.
Thus if $L1 < L2 < U1 < U2$ then

```
power(a(r), x, r, L1, U1) +
      power(b(r), x, r, L2, U2) =
power(a(r), x, r, L1, L2-1) +
      power(a(r)+b(r), X, r, L2, U1) +
          power(b(r), x, r, U1+1, U2)
```

If the dummy variables (r) of two power functions are different, then the second name is replaced by the first in the addition, provided the latter does not appear in the first parameter of the second function. If it does, the second name will replace the first.

Example

```
power(r+1, x, r, 1,INF) +
      power(p+3, x, p, 100, INF) =
power(r+1, x, r, 1, 99) +
      power(r+4, x, r, 100, INF)
```

However, if both conditions occur, for example in the case

```
power(r*s,x,r,1,10)+power(s**r,x,s,5,20),
```

then a message is typed at the console, requesting the

user to change the summation variable in one of the functions.

The bounds of the summation, L and U, must be integer valued. '%addsum' also searches for the recognised function 'sigma', of the form

sigma(EXPR(r),r,L,U)

which represents the summation of the term EXPR(r) for r=L,1,U. The rules described for 'power' are also applicable to the summation of instances of 'sigma'.

6) %diff LABEL EXPR LIST %wrt NAME %into LABEL EXPR LIST

This gives as a result, the derivative with respect to NAME, of each operand given by the first LABEL EXPR LIST. The rules of differentiation that are applied are given below, where S is the operand and R the result of differentiating with respect to x. f'(x) is the derivative of f(x) with respect to x, and is evaluated if possible.

a) S = a

R = 0

b) S = x

R = 1

c) S = a * f(x) ** n

R = a * n * f(x) ** (n - 1) * f'(x)

d) S = a ** f(x)

R = a ** f(x) * log(a) * f'(x)

e) S = log(f(x))

R = f'(x) / f(x)

```
f)  S = exp(f(x))

    R = f´(x) * exp(f(x))

g)  S = sin(f(x))

    R = f´(x) * cos(f(x))

h)  S = cos(f(x))

    R = -f´(x) * sin(f(x))

i)  S = tan(f(x))

    R = f´(x) * (sec(f(x))) ** 2

j)  S = cot(f(x))

    R = -f´(x) * (cosec(f(x))) ** 2

k)  S = sec(f(x))

    R = f´(x) * tan(f(x)) * sec(f(x))

l)  S   cosec(f(x))

    R = -f´(x) * cot(f(x)) * cosec(f(x))

m)  S = f(x) * g(x)

    R = f´(x) * g(x) + f(x) * g´(x)

n)  S = f(x) / g(x)

    R = (g(x) * f´(x) - f(x) * g´(x)) / g(x) ** 2

o)  S = power(f(r), x, r, L, U)

    R = power((r+1) * f(r+1), x, r, L-1, U-1)
```

Examples

```
    S = 4 * a * x ** 3

    result = 12 * a * x ** 2

    S = x ** 3 * (a + 3 * x) ** 2

    result =  3 * x ** 2 * (a + 3 * x) ** 2 +

              6 * x ** 3 * (a + 3 * x)

    S =       power(r ** 2, x, r, 0, 100)
```

```
result =  power((r + 1) ** 3, x, r, -1, 99)
```

7) %subs FORCL INTOCL

The FORCL has two forms

EX/LB %for EX/LB

EX/NM/LB

where EX/LB represents an expression or a label expression, and EX/NM/LB represents an equation, a name or a label expression. Thus the alternatives are

```
%subs P %for Q %in R          (i)
%subs Q = P %in R             (ii)
%subs N %in R                 (iii)
```

In the first two cases, each instance of Q in R is replaced by P. In the third, N is the name of a scalar variable or an array element. The value of N replaces its name in R.

Examples

```
a = s + t + u * s + t
%subs p + r %for s + t %in a
result = p + r + u * s + t
```

Notice that %subs takes notice of the precedence of operators, and so the second occurrence of ´s + t´ cannot be replaced.

```
a = s * t * u * v
%subs s * t = q + r %in a
result = (q + r) * u * v
a = s + v + u * (s + t)
s = p * q
```

%subs s %in a

result = p * q + v + u * (p * q + t)

If N is one of the reserved words PI or EXP, it it replaced by its numerical value, 3.141593 or 2.718281.

8) %print %results LABEL EXPR LIST (a)

%print %results (b)

%print %no %results (c)

Generally the results given by any algebraic command are printed out at the terminal as the commands are obeyed. If '%print %no %results' is given as a command, this facility is suppressed for any commands executed after it. The printing may be 'turned on' again by the command '%print %results'.

It is possible that the results given by some commands are needed while others are intermediate results which should be suppressed. By placing the two classes in different blocks, this effect can be achieved, since '%print %results LABEL EXPR LIST' causes the results which are to be put into the blocks listed to be printed, while any others are suppressed. The effect of this command continues until it is changed by (b) or (c). A further command of the form (a) adds the values in the LABEL EXPR LIST to the values previously given, but does not delete any.

Example

1:1 (a + b) * (c + a)

1:2 (3 + a) * (a - 1)

```
1:3      (3 + a + c) * (a - 1 + b)
         %print %results 3
1:4      %distrib 1:i %into 2:i
1:4:1    %simplify 2:i %into 3:i
         %do 1:4 %for i = 1,1,3
3:1      a * c + b * c + a ** 2 + a * b
3:2      -3 + 2 * a + a ** 2
3:3      -3 + 2*a + a ** 2 - c + 3*b + c*a + a*b + c*b
```

Here we have suppressed the printout of block 2, which receives the results of ´%distrib´.

5.14 COMMAND LISTS

A number of commands may be given in the same statement. E.g.

%simplify 1:1,:3, %expand 2:1

If a command A is not the first of a COMMAND LIST, and is algebraic it may have as its operands the results given by the algebraic command B which precedes it. This is specified by ommitting the operand list of the command A. Hence the statement is of the form

B INTOCL, A %into LABEL EXPR LIST.

If command A is a general command and not the first of a command list, then it may use the same set of operands as the general command B that precedes it. Hence the statement in this case is of the form

B LABEL EXPR LIST, A

E.g. %simplify 1:1,:3 %into 2:1,:2, %expand %into 3:1,:2

```
%write 2:1,3:7, %do.
```

In 5.2 we discussed the use of condition loops with the command '%do'. We may now generalise those remarks, and allow a condition loop to be placed after any list of commands, to give a statement of the form

COMMAND LIST CONDITION LOOP .

All the commands of the list are executed before the condition is checked, and the commands are re-executed if required.

For example,

```
%do 1:1, :2 %for i = 1,1,3
```

causes blocks 1:1 and 1:2 to be executed three times.

```
1:1     i = i + 1
:1:1    .....
:2      .....
        .....
2:1     a * 3 * b + 5 * a * b
:2      6 * a - 3 * a
        ........
        i = 0
        %do 1:1, %simplify 2:i %into 3:i %until i=10
```

At the head of the block 1:1, i is increased, the rest of the block is executed, and then the contents of 2:i have the command %simplify applied to them. Next the CONDITION is tested, and the list is re-executed until it is satisfied, thus applying the %simplify command to blocks 2:1, 2:2, ... 2:10.

```
4:1        .....
:1:1       .....
:7         j = j + 1
           j = 1
           %write 3:1,%do %for i = 1,1,5
           %do 4:1 %while A(j) >= 0
```

The contents of blocks 3:1 to 3:5 are written at the terminal and then executed. Next block 4:1 is executed provided that A(j) >= 0. j is increased in block 4:1, so the elements of array A will be processed in ascending order until a negative quantity is reached.

5.15 Patterns

When one is dealing with algebraic formulae, it is often the structure of the formula that one is interested in, and so it was decided to supply pattern matching facilities in AML. These take the form of conditions, known as pattern conditions. There are two forms:

A %matches PATTERN (a)

A %contains PATTERN (b),

where A is of the form NAME LIST? and defines a variable or array element. (a) can be defined as follows

1) If PATTERN is a name or constant, then A must be that same name or constant for the condition to be satisfied.

2) If PATTERN is the key word

   a) %integer, %real, or %rational

   b) %numeric

c) %algebraic

d) %any

e) %factor

f) %term

then A must be

a) a constant of the correct type

b) a numeric constant

c) an algebraic constant or function

d) any expression.

e) an operand or an expression whose main operator is **.

f) an operand, or an expression whose main operator is *, /, or **.

3) If PATTERN is of the form

Q op R

where op is any operator, then A must be of the form

G op H

where G matches Q and H matches R, and op is the same operator as in PATTERN.

4) If PATTERN is of the form P(Q1,Q2,...Qn) and P is '%name' or is a name, then A must be of the form F(G1,G2,...Gn) where F matches P and Gi matches Qi for i= 1, 2, ...n.

(b) is said to be satisfied if there is a sub-expression B of A such that

B %matches PATTERN.

Using these two conditions it is possible to set up tests

for looking at the structure of a formula.

Example

```
a = b + c - d

%if a %matches %any - %any %then %do 1
```

This tests whether the value of 'a' is an expression that has a minus sign.

The precedence of operators is taken into account during pattern matching. Hence

```
a * b - c %matches %any - %any      is true
a * b - c %matches %any * %any      is not true
a * b - c %contains %any * %any     is true.
```

5.15.1 Extractors

Once it has been determined that a formula matches a particular pattern, it may be useful to pick out the parts of the formula that correspond to certain parts of the pattern. This can be done by using an EXTRACTOR which takes the form

```
NAME _
```

and prefixes the part of the pattern that is to be considered.

For example,

```
1:1     a_%integer + %any
:2      b_(sin(%integer) + %any) + %any
:3      f(c_%any) + %any
```

When the match has been established a pointer is set up from the extractor name to the part of the formula that it identifies. Thus

```
d = 6 + p * q
%if d %matches 1:1 ...
```

sets ´a´ pointing to 6.

Similarly

```
m = sin(30) + 2 + p * r
%if m %matches 1:2 ...
```

sets ´b´ pointing to sin(30) + 2.

and

```
n(1) = f(x + y) +p * t
%if n(1) %matches 1:3
```

sets ´c´ to x + y.

Extractors may be used to change an algebraic expression that is the value of a variable. Once the link has been established, the change is made by a statement of the form

```
NAME <- EXPR          (A).
```

NAME must be an extractor name that is linked to an algebraic expression. Then the part of the expression that is linked is replaced by EXPR.

E.g.

```
a <- 2 * r
```

changes d to

```
2 * r + p * q
```

Similarly

```
b <- 5
c <- z ** 2
```

give

```
m = 5 + p * r
n(1) = f(z ** 2) + p * t
```

Extractors may also be used with '%contains'. If there is more than one instance of the pattern occurring in the formula, the left most one is used in establishing the link. E.g.

```
z = a * b * c * d
%if z %contains %any * q_%algebraic %then ...
```

is satisfied, and q points to b.

5.16 Selectors

When one is handling long algebraic formulae, it is often desirable to pick out part of an expression to examine it more closely. This may be done using patterns and extractors. The SELECTORS supply another method.

If an expression (bracketed) forms an operand of an expression, then it may be prefixed by a SELECTOR. Then the selector will pick out part of the bracketed expression, and this part will be used in the evaluation of the outer expression.

Example

```
6 + (SELECTOR 1:2) + c          (A)
```

will take the required part of statement 1:2, and use this to evaluate (A). The simplest SELECTORs are

```
%rhs      and      %lhs      (1)
```

Thus for

```
1:2      x + y = z - 3,
```

if SELECTOR = %rhs, (A) becomes

```
6 + z - 3 + c
```

when evaluated.

If SELECTOR = %lhs, (A) becomes

6 +x + y + c.

SELECTOR may also have the form

DESCRIPTION %of (2)

where

DESCRIPTION = N %th PATTERN or PATTERN.

PATTERN is described in 5.15, and N is an integer valued expression.

Thus for

1:2 x + 3 + y - 7,

and SELECTOR = 2 %th %integer,

(A) becomes

-1 + c when evaluated.

If the second form of DESCRIPTION is used, the first such PATTERN of the expression is used. SELECTOR may use itself recursively, in the form

DESCRIPTION %of SELECTOR (3).

Examples

4 %th %integer %of %rhs 1:3

(m + 1) %th %opd %of 2 %th %term %of 2:1

The search for selectors works backwards, so that these examples would have the meaning one would naturally attribute to them. So in the second example, the second term of 2:1 is found, and then the (m+1)-th operand of that is found.

VI Implementing AML - The Storage Tree

There are two basic ways of processing a program written in a high level language, to enable a computer to obey it. The first is a two step process. A compiler translates the program into machine code, and then this machine code is loaded and obeyed. The second way is to use an interpreter which discovers the meaning of a statement, obeys it, and then proceeds to the next. Most batch systems use compilers because they are generally more efficient than interpreters. However when a system is designed to be used in desk calculator mode, an interpreter is more usual. The advantage of a compiler lies in the fact that the statements, although they are obeyed several times, need only be translated once. In desk calculator mode, where the statements are lost after execution, this advantage no longer exists.

Since the unlabelled statements of AML are used in desk calculator mode, an interpreter is used for them. For the labelled statements, three choices are available

1) The statement could be translated into a series of machine code instructions, and these could be stored.

2) Some intermediate stage, such as an analysis record could be stored.

3) The statement could be stored unaltered, and interpreted in the same way as an unlabelled statement when it is accessed.

If a statement is accessed many times, the advantage of repeating as little of the translation process as possible is obvious. However another consideration makes this line of attack impossible. In any translation process, the first thing to do is to decide what kind of statement is being handled. But AML differs from most languages in that a statement need not be an executable statement. Hence until the statement is accessed, the form it is expected to take is not known. This situation could be overcome (although in a very time consuming manner) if each statement that could possibly form part of an executable statement had a distinct syntax. This is not so. There may be two or more syntax definitions that a statement may satisfy, and until the statement is accessed it is impossible to know which one is required. In other words, it is not context free. For example the statement

```
1:1    a+b*c
```

could be recognised as an expression or as a pattern. Therefore until the statement is accessed, the phrase name it must be matched against is not known. For this reason the third method must be chosen. It has the additional advantage that the same process is applied to both labelled and unlabelled statements.

6.1 The Storage Tree

The next problem to be considered is the method of storing the labelled statements. The block structure of the language obviously suggests a tree. Let us consider the

statements

1:1 a

1:1:1:3 b

2:1 c

2:3 d

4:2 e

4:1:1 f

1:2:3 g

3:1:2 h

A tree structure which would store these statements is given by Fig. 1

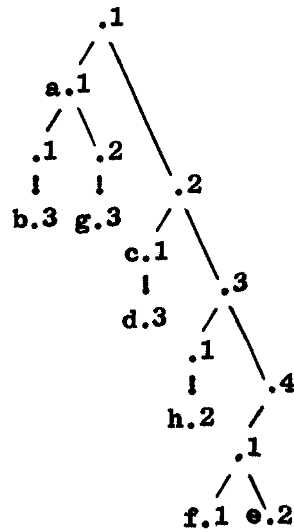


Fig. 1

These can equivalently be represented by the nested list structure given in Fig 2.

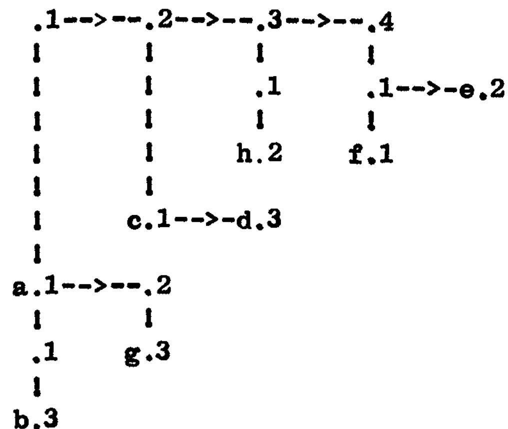


Fig. 2

To represent a node (e.g. .1) of the list, we require three values. The first, NO, contains the number of the node. I.e. the number 1 for the node '.1'. The second, BELOW, points to the node below it, and the third, AFTER, points to the node after it.

Fig. 3 reproduces Fig. 2, but in it each of the nodes has been given a name. Notice that capital letters name the node; small letters are used for the statements at certain nodes.

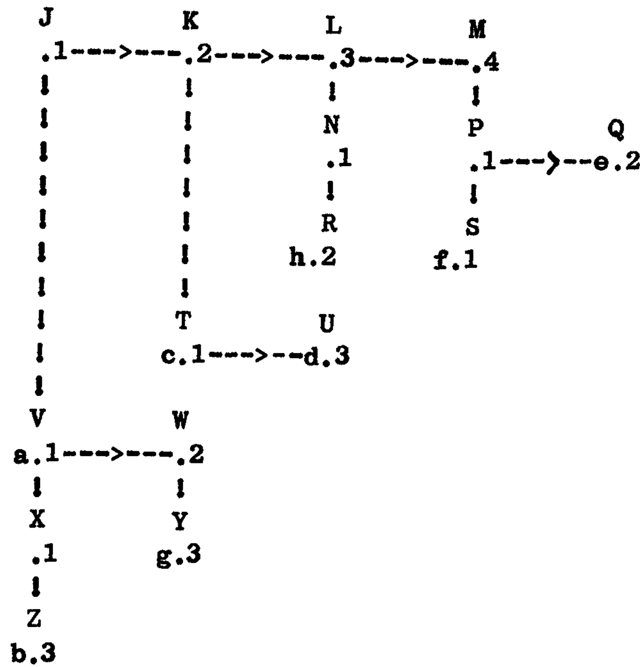


Fig. 3

Let us consider the node at J. This has the value 1, and so NO(J)=1. The node below J is V, so BELOW(J)=V. Similarly the cell after J is K, so AFTER(J)=K. The node T does not have a node occurring below it, and so in this case BELOW(T)=0. If a node has no node after it, the situation is more complicated. Consider the node at Z. Now NO(Z)=3, but the label actually represents the label 1:1:1:3. To find label 1:1:1:3 in the tree is straightforward. First label 1 is found at J. V=BELOW(J) gives 1:1, X=BELOW(V) gives 1:1:1, and Z=BELOW(X) gives 1:1:1:3. However, to find out what label Z represents is not possible at the moment. To enable the required back tracking to take place, AFTER(Z) is set to -X, AFTER(X) is set to -V, etc. In general the value of AFTER for any cell that comes at the end of a list is -1 multiplied by the position from which the list hangs. To

take another example, AFTER(U)=-K. U does not hang directly from K, but it comes after T, which is below K. Fig. 4 gives the representation of the complete tree, illustrating the contents of the three divisions. STATE contains the statement at the node, if there is one.

NO B A S

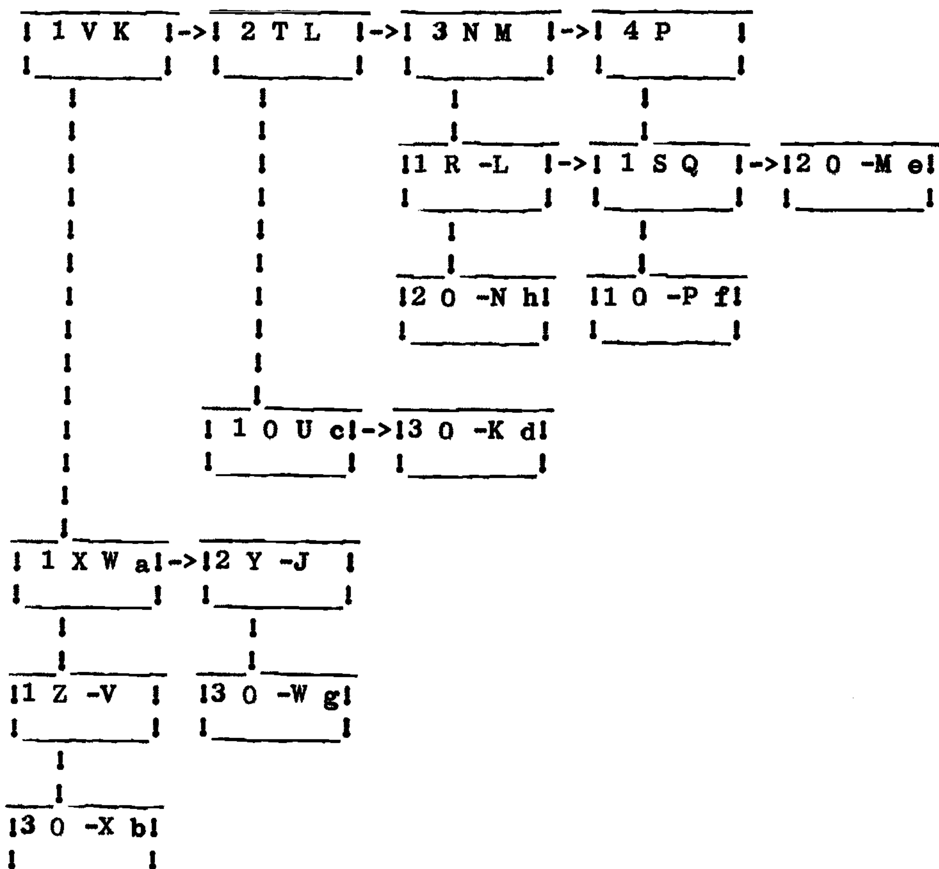


Fig. 4

where B stands for BELOW

A stands for AFTER

S stands for STATE

The internal representation of this structured list is formed by three arrays, NO, BELOW, and AFTER. NO is a byte integer array, since the value of a particular component of a label cannot exceed 255. BELOW and AFTER contain pointers

to other nodes of the tree, and so are short integer arrays. The size of the arrays is stored in an own integer, so that it can be easily altered. At present the arbitarily chosen value of 300 is used for the size of the arrays.

Notice that in the diagram AFTER(M) is undefined. This is because the tree has not yet been given any fixed values. NO, AFTER and BELOW can be regarded as cells of a list. The k-th cell of the list is composed of NO(k), BELOW(k) and AFTER(k). These cells are added to the tree whenever a new label is given. The integer asl is a pointer to the next free cell of the list, and is initially 2. The whole tree is hung from cell 1, which has dummy values in NO and AFTER. So, in Fig. 4, AFTER(M) would be -1. pointing back to the head of the tree.

6.2 Storing the Statement

The method described above has set up a tree representing the structure of labelled statements of the program. Using the free list new labels can easily be inserted into the structure. Also the statement stored at a particular label may be associated with the number of the cell representing that label. Since statements are to be stored exactly as they are presented, a byte integer array may be used to hold it; each byte holding one character of the statement. The easiest and quickest way of storing the statements is to use a two-dimensional array of the same length as the free list of cells described above.

Example

If the statement

1:2   i=3

was presented, the structure would be as follows.

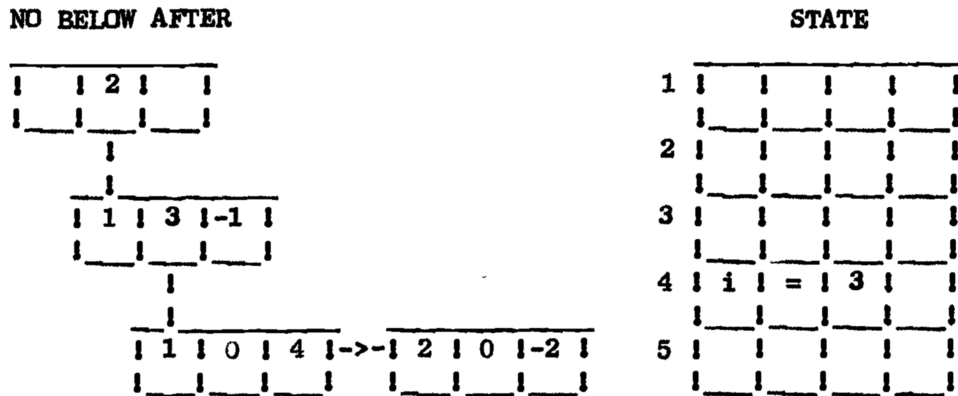


Fig. 5

Now it must be possible to allow for long algebraic expressions of about 300 or 400 characters, but the majority of statements will be much shorter, probably of less that 20 characters. (See Appendix B) It will be seen therefore that this method uses a large amount of space, much of which will not be used.

This program was first developed on the English Electric KDF9, with a core store of 16 K of 48 bit words. It soon became apparent that space was at a premium, and so the ability to access a statement directly had to be sacrificed, in order that it could be stored more compactly.

To save space, all the statements presented to the system are stored in one byte integer array called ALG. An array of pointers is used to access the statements. Thus the statement associated with the cell k is stored in the array from ALG(i) to ALG(j), where i and j can be obtained from

the value of k. The array ALG is used for storing different kinds of information; in fact any string of information whose individual elements can be stored in a byte is stored in ALG. The structure that is set up to deal with this will be described in Chapter 9.

6.3 The Internal Representation of a Statement

Before a statement is stored in ALG, two alterations are made to it, both designed to save space. There is a large number of key words in AML, some of them quite long. Each of these can be associated with a number, and this number, referred to as the key number, replaces the key word in ALG. Key numbers may lie in the range 129 to 254. In fact there are 63 key words in AML, thus giving an actual range of 129 - 192. The ISO representation of a character has a value less that 128, and so key numbers can be stored in a byte integer, without being confused with characters.

Names are also replaced by a number in ALG. Each time a name is presented to AML, a dictionary is consulted, and the name is inserted if it is not already there. Thus each name has a number associated with it, which is its position in the dictionary. 256 names, numbered between 0 and 255 may be entered into the dictionary. In order to avoid confusion between the dictionary number of a name and the ISO representation of a character, the former is preceded by the character ′%′ which can not occur anywhere else in this representation of the statement. (Since key words are not stored in their external form.) This means that if a name

consists of a single character an extra space will be used. However it was felt that the average length of a name would be greater than two, and so an overall saving would be made.

Example

The statement

%result = alpha + beta

is stored as

153 ´=´ ´%´ 1 ´+´ ´%´ 2

where 153 is the key number of ´%result´, and 1 and 2 are the dictionary numbers of alpha and beta respectively.

6.4 The Dictionary

Hash coding is used to insert a name in the dictionary. This means that although 256 names may be used, the efficiency of the look up will drop if more than 80% of the positions (i.e. about 200 names) are used.

The formula for the hash coding is fairly standard: the first four characters of the name are treated as an integer, and the remainder on division by 251 (the highest prime below 255) is taken as the hash code. If the name has less than four characters, the extra positions to the right of the characters are filled with zeros.

The dictionary consists of two parts: a byte integer array NAME, in which the actual names are stored, and a short integer array ALPHA which is accessed by the hash coding. NAME is of size 1000, thus assuming an average length of five characters for 200 names. ALPHA is of course in the range 0-255.

When a name is presented to AML, its hash code, h, is calculated, and the position in ALPHA given by h is examined. If it is empty, the name has not been used before. Therefore it is copied into NAME at the first empty location, preceded by its length, and ALPHA(h) is set pointing to the the entry in NAME containing the length. If ALPHA(h) is full, the name pointed to is matched against the name presented. If they are the same, the process stops. If not, ALPHA(h+1) is examined (and the value of the hash code of the name is now h+1). This process continues until an empty ALPHA is found, or until the name is recognised. If the dictionary is practically full, this will result in a cyclic search of ALPHA. The program is faulted if there is no more space in ALPHA for an unrecognised name.

VII Implementing AML - Analysing a Statement

The interpreter runs in a continuous loop, stopping only when it encounters the command ´%finish´, which indicates that the program is ended. The steps of the loop are

1) Read in next statement.

2) Go to (6) if it is unlabelled.

3) Find position of the label in the Storage Tree.

4) Store the statement in ALG, associating it with the position found by (3).

5) Go back to (1).

6) Obey statement.

7) Go back to (1).

Steps (3) to (5) have been discussed in Chapter 6. The first decision to be made about step (6) was whether or not to apply a syntax analyser to the statement. An attempt was made to process the statement as it stood, but was found to be inpracticable, as so many tests were required to establish whether or not a statement was acceptable. Thus step (6) is divided into two parts:

a) Get analysis record of the statement.

b) Process analysis record.

The syntax has been decided during the design of the language, and it remains to formalise it, and to obtain a form which could suitably be stored in the interpreter. This syntax is given in Appendix A. Once the statements of the language have been defined, it is necessary to consider

three things.

1) The writing of the formal syntax of AML, so that a potential user could easily refer to it.

2) The production of a representation of the syntax suitable for storing in the machine.

3) The writing of a syntax analyser routine to produce an analysis record for a given statement.

The syntax of both IMP and AML consist of a list of PHRASE NAMES, each of which is given a definition. A phrase name is defined by a list of ALTERNATIVES, each of which has a number of ITEMS. An item may be either a LITERAL, or the name of another phrase name.

Example

(CONDITION) = (SC)´%AND´(ANDCL)!

(SC)´%OR´(ORCL) ! (SC)

Here the phrase name (CONDITION) is defined by three alternatives. The first two each have three items, and the last has one. Each of the first two alternatives has a phrase name as its first item, a literal as its second, and a phrase name as its third. This means that a (CONDITION) may be either

1) An instance of the phrase name (SC), followed by the word ´%and´, followed by an instance of the phrase name (ANDCL).

2) An instance of the phrase name (SC), followed by the word ´%or´, followed by an instance of the phrase name (ORCL).

3) An instance of the phrase name (SC).

7.1 A Comparison of the IMP and AML Syntax Analysers

The syntax analyser of IMP is highly recursive, and requires a certain amount of back tracking. The following is an algorithmic description of it, comparing text stored in the array T between m and n against a phrase name ´phr´. The call of the analyser is a function which gives the result 1 if T is an instance of phr, and 0 otherwise. The declaration of this function is

```
%integerfn analyse(%integerarrayname T,%integer m,n,phr).
```

1) Get the first alternative.

2) Get the first item of this alternative.

3) If the item is not a literal, go to (6). Match the literal against the next character in the text. Go to (5) if they do not match.

4) Result = 1 (success) if this was the last item. Result = 0 (failure) if the end of the text has been reached. Get the next item. Go to (3).

5) Result = 0 if this was the last alternative. Get the next alternative. Go to (2).

6) The item is a phrase name. Call Anal for it. Go to (4) if it gives a result 1, and to (5) for a result 0.

This describes a version slightly different from that actually used, in order to demonstrate the method clearly. Let us consider what happens when the text ´x>1´ is tested against the phrase name (CONDITION).

1) The first item of the first alternative is considered, i.e. (SC).

2) Analyse is called for (SC), and the result is 1.

3) This item was not the last, so we go on to process the next item. However since the end of the text has been reached, this alternative will not do.

4) The first item of the second alternative is considered. Again it is (SC) which gives a result 1 for analyse. However there is no text to match ´%or´, so this alternative fails for the same reason as the first.

5) The third alternative is considered. The first item is (SC) which gives a result 1 for analyse. Since this is also the last item, the result 1 is returned.

In the course of doing this recognition, analyse has been called three times for the phrase name (SC). This can be improved by changing the definition of (CONDITION) to

```
(CONDITION) = (SC)(RESTOF COND)
(RESTOF COND) = (SC)´%and´(ANDCL)!
´%or´(ORCL)!(SC)
```

The syntax of IMP is arranged to minimise back tracking. In order to take as little time as possible, the most common alternatives of a particular phrase name are put first. Unfortunately this does not always reduce the work done. For example consider the phrase name (SS) (source statement) in IMP. Its most common alternative is a (UI)

(unconditional instruction) which has twelve alternatives. This means that any (SS) which is not a (UI) must be tested against all twelve alternatives of (UI) before the next alternative of (SS) is examined. Consequently a fair amount of testing must be done before an alternative even halfway down the list is recognised.

AML approaches the problem from a different angle. Of the two kinds of items that occur, a literal is much simpler to test. Hence the analyser first looks for any literal expected in an alternative, and if these match, then the phrase names are tested. Let us reconsider the definition of the phrase name (CONDITION). The definition is the same in AML as in IMP, i.e.

(CONDITION) = (SC) '%and' (ANDCL)!
(SC) '%or' (ORCL)!(SC)

If the text 'x>1' is tested against this phrase name, the steps taken are as follows.

1) Find the first literal of the first alternative of (CONDITION). This is '%and'.

2) 'x>1' does not contain '%and', therefore move to the second alternative.

3) The first literal of this alternative is '%or'. 'x>1' does not contain this literal, so move to the next alternative.

4) This alternative does not contain any literals, so consider the first phrase name,(SC), which is tested against 'x>1'.

5) The result obtained is 1, and there is no other item in the alternative. Therefore ´x>1´ has been recognised as the third alternative of (CONDITION).

Using this method, the test of (SC) against ´x>1´ is only done once. In general if alternatives containing literals are listed first in the definition of a phrase name, the quick test for literals is sufficient to dismiss many cases, and the recursive call on Analyse is not needed.

Thus the method of approach is as follows.

1) Get the first alternative of the definition.

2) Search it for a literal.

3) Match the characters of the text against this literal L. Go to (6) if there is no such L in the text.

4) If there are items before L, or if no literal was found in SYN, call Analyse for each item of the alternative. Go to (6) if any of these calls fails.

5) Result is 1 (success) if there was no literal, or if the literal was the last item of the alternative. Otherwise search the remainder of the alternative for the next literal L, and go to (3).

6) Result is 0 if this was the last alternative. Get the next alternative, and go to (2).

More code is needed in the more sophisticated analyser, and so saving time, as so often happens, requires an increase in space. Appendix C gives some figures measuring

the difference in performance of the two methods.

7.2 Lists and Null Expressions

In describing any syntax, it is frequently necessary to describe a list of some entity, for example a list of names, a list of array bounds, or a list of formal parameters. The general form of the description is always the same. Let (NLIST) define a list of instances the phrase name (N). Then the definition of (NLIST) is

(NLIST) = (N) ´,´ (NLIST) ! (N)

Because this form is so common, it was decided to adopt the convention that a ´*´ occurring after any phrase name indicates that a list of that phrase name is allowed, so that (N)* = (NLIST). For example, an array declaration is defined as

´%array´ (ARDEFN)*

that is, the literal ´%array´ followed by a list of (ARDEFN)s.

In several places there are also items which may or may not be present. In other words they are allowed in the alternative, but their absence does not cause the alternative to be rejected. To indicate this property these items, which may be literals or phrase names, are followed by a ´?´. For example, the assignment statement is defined as

´%real´?(NAME)(LIST)?´=´(EXPR).

Examples of assignment that fit this definition are

x = 1

```
%real y = 1
p(3) = z
%real q(a) = x
```

A list of phrase names may also be followed by a question mark.

For example,

```
'%do' (LB)*?        (i)
```

then

```
%do 1:1
%do 1:1,2:3,4:5:1
%do
```

are all examples of (i).

7.3 Built in Phrases

The discussion so far describes and justifies the syntax given in Appendix A. Also a rough outline has been given for the approach to part (3) of the tasks that have been set. Before considering part (2), a brief discussion on built in phrases is needed.

A phrase is said to be 'built in' if the syntax analyser is not applied to is in the usual way, but instead special steps are taken to recognise it. For an example, let us consider the phrase (NAME). A (NAME) in IMP or AML is defined as a letter followed by a string of digits or letters. This informal statement defines the phrase to the user. However to define it following the formal rules of syntax is quite tedious.

```
(NAME) = (LETTER)(RESTNME)*?
```

(RESTNME) = (LETTER) ! (DIGIT)

(LETTER) = 'a'!'b'! ...'y'!'z'

(DIGIT) = '0'!'1'! ... '9'

Moreover there is a much quicker way of deciding whether a piece of text is a letter than to apply the formal definition. If t is an integer in IMP, then the condition

'a'<=t<='z'

is true if t represents a letter, and false otherwise. Hence it is possible to make (NAME) a built in phrase, and write some code using the test described above to decide whether the text being considered is a (NAME). This is what happens in IMP. In AML the situation is different. The input to the analyser is not the source text, but the modified statement that was described earlier. Hence to establish whether the text is a name, it only necessary to check that it is of the form '%'n, where n is any number.

There are eight built in phrases in AML; the reasons for choosing these particular phrases to be built in are given later.

7.4 The Stored Form of the Syntax.

The form of the syntax which can be stored in AML is also derived from IMP. A short array, SYN, is used to hold the information. Let the phrase P be defined as

(P) = I1 I2 I3 ! I4 I5 I6 ! I7 I8 I9

Fig. 1 describes the entries in the array for this definition.

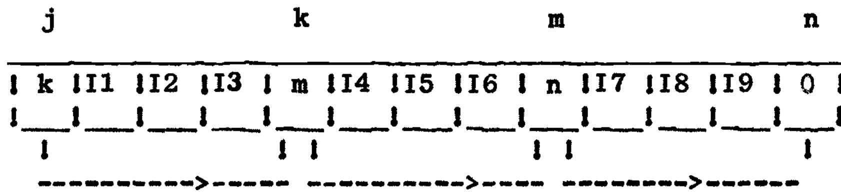


Fig. 1

The description of P begins at j. SYN(j) points to k, the location after the last item of the first alternative. K points to the end of the second alternative, m, and m points to the end of the third alternative, n. Since this is the last alternative, SYN(n)=0.

Now let us consider the items. These can either be literals or phrase names. For the former the ISO value of the character is stored, and for the latter a pointer to the beginning of its description. However we must recall the alterations that were made to the statement as it was read in. Key words are replaced by their key numbers, which are in the range 129 to 255. Hence literals, which may be ISO characters or key numbers, lie in the range 10 to 255. To distinguish between pointers to phrases and literals, it is therefore necessary to make the pointers have values outside this range. The pointers all refer to locations in SYN, so the array was chosen to have bounds from 256 upwards.

Two minor points remain. ´*´ and ´?´ must also be represented in SYN, but cannot have their ISO code values. So -1 was chosen to represent ´*´, -2 for ´?´, and -3 for the combination ´*?´. Secondly, built in phrases are not described in the syntax, and so cannot be represented by

pointers to SYN. Since 10 is the smallest number that represents a character, the numbers one to eight are used to represent the built in phrases.

Example,

The alternative of (ACTIVEST) that defines a routine declaration is

´%local´? (RT) (NAME) (FPP)? ´%at´ (LB)

This is represented in SYN as

130	-1	433	1	438	-1	136	5

where

130 is the key number of ´%local´

-1 represents ´?´

433 is the position in SYN of the definition of (RT)

1 represents the built in phrase (NAME)

438 points to the definition of (FPP)

-1 represents ´?´

136 is the key number of ´%at´

5 represents the built in phrase (LB).

7.5 The Routine Analyse

Each statement that is read in to the system is processed by the routine called Analyse. Using the information contained in SYN, it processes the text held in the array AR, between positions m and n. If it recognises the text, it puts the analysis record of AR in PAR, and returns with result 1. Otherwise the result is zero. ´Ptr´

gives the position in PAR at which the analysis record ends. 'phr' is the number of the phrase name that is to be matched against AR. Thus from the main body of the program, the routine is called with phr set to the phrase number of (ACTIVEST). However phr may take the value attributed to any accessible phrase name. The analysing itself is done by the function Anal which is local to Analyse. Its parameters m, n, and ptr, correspond in meaning and type to the parameters of the same name in Analyse. The integer parameter i points to the beginning of the description of the syntax in SYN, except for built in phrases, when i is the phrase number. We will discuss the general case first. Built in phrases and the handling of '*' and '?' will be described later.

To make the process easier to follow, we will examine how Anal deals with a particular statement. The example chosen is

```
%do 5:3 %unless i = 7
```

which is to be tested against (ACTIVEST). On entering Analyse, j is set to LINE(i), which points to the beginning of the phrase name in SYN. The variable p (global to Anal) is set to m, and anal(j,m,n,ptr) is called.

In Anal, SYN(i) contains a pointer to the place where the second alternative begins, and so the locations between i and SYN(i) must be examined.

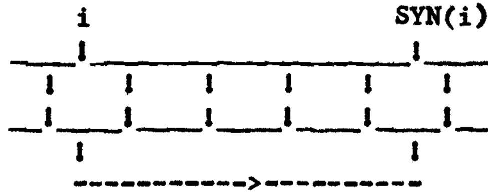


Fig. 2.

Thus Anal cycles through the locations i+1 to SYN(i)-1 to determine whether any of the entries here are literals. (Values for literals are between 10 (newline) and 255.) Let us suppose that SYN(j) contains a literal. Now the routine must cycle from m to n to see if AR contains that same literal. If it does not, this alternative does not match AR. Hence i is set to SYN(i), and the process is repeated. If SYN(i)=0, however, no more alternatives are available, and so the result is zero.

Returning to our example, suppose the first alternative of (ACTIVEST) is

(UNCONDST) ´%if´ (CONDITION)

This is represented in SYN as

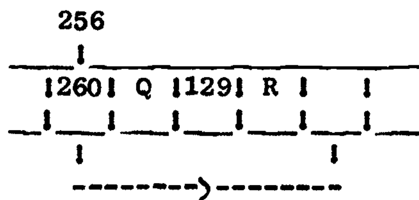


Fig. 3.

where Q points to the description of (UNCONDST), and R points to the description of (CONDITION). At SYN(258) we find the number 129, which is the key number of ´%if´. This does not occur in the statement we are processing so we pass

on to the next alternative.

If the literals do match the beginning of statement should be processed. (Assuming that the key word is not at the beginning.) Let the literal in AR(k) be the same as that in SYN(j). Then anal(syn(i+1),m,k-1,r1) is called, where r1 is a pointer to the array PAR. (A different variable from ptr is used in case some part of the statement is not recognised.) If Anal returns the result zero, we move on to the next alternative. Otherwise p will point to the last element of AR that was recognised. In the example, let the second alternative of (ACTIVEST) be

(UNCONDST) ′%unless′ (CONDITION).

The representation in SYN is

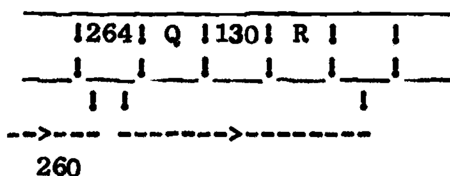


Fig. 4

The statement we are examining contains the literal ′%unless′, and so the first part of the statement, i.e.

%do 5:3

is examined. Hence anal(Q,m,k-1,r1) is called, and returns a result one, since ′%do 5:3′ is recognised as an (UNCONDST). P points to k. In the example we have only one entry between i and j. In general, however, Anal is called for each cell between them, provided that in each case the result from the previous call was 1. M is set to p before

each call.

After the last call, p must point to k, since there can be no unrecognised part before the literal. Then m is reset to the position after the literal, i points to j+1, and we return to the beginning of the process.

If the cells between i and SYN(i) have no literals, Anal must be called for each entry of the alternative. Thus in the example, anal(R,k+1,n,r1) will be called. When all the cells have been used, p is set to the position after the last recognised cell of AR. There is no need for the whole statement to be recognised at this stage since another call of Anal may deal with the end of it. However the whole statement may not be recognised before all the necessary calls of Anal have been made. When control eventually returns to Analyse, the result of that function is 1 provided that the result of Anal is 1, and that p=n+1.

It is possible to have more than one (ACTIVEST) in one statement, provided that they are separated by semi-colons. Hence if a statement has been recognised as an (ACTIVEST) and if p<n+1, with AR(p)=´;´, Anal is called again for (ACTIVEST), for the rest of AR. Then when each (ACTIVEST) has been identified successfully, PAR(1) will contain the number of (ACTIVEST)s there are in the statement. This is permitted only for the phrase name (ACTIVEST), and for no others.

7.6 The Analysis Record

So far the discussion has shown how Analyse determines whether a statement belongs to the syntax of AML, without any reference to the analysis record that is set up. This is contained in the array PAR, and the final result is of the form given by Fig. 5.

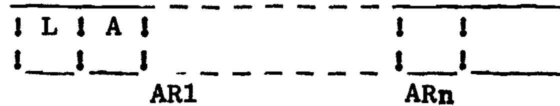


Fig. 5

where L is the length of the record, and A is the number of the alternative, and AR1 ... ARn are the analysis records of the phrase names of the alternative.

When Anal is first entered from Analyse, the name type parameter ptr has been set to 2. This is the pointer to PAR. If there is only one alternative of the phrase name, A is ommitted. In this case r1=ptr+1, otherwise r1=ptr+2. Hence when Anal is called with r1 as the name type parameter, another pointer to PAR is set up.

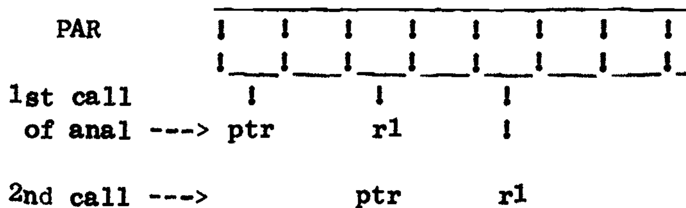


Fig. 6

As an example, let us consider the alternative

(IU)(CONDITION) ´%then´ (ELSECL)

of (ACTIVEST). This will be matched against the statement

%if i = 0 %then %do 3

The literal ´%then´s match, and so Anal is called for (IU), whose definition is

´%if´ ! ´%unless´.

At this stage the pointers to PAR are in the position given by Fig. 6.

The literal ´%if´s are matched, and since this ends an alternative of (IU), Anal returns with result 1. Before it does so, however, PAR must be filled in. The position PAR(ptr) is filled with the length of the analysis record of the phrase (i.e. r1 - (ptr + 1)). The position PAR(ptr+1) holds the number of the alternative that was recognised. Ptr is set to r1 [illegible], and the function returns. These values are filled in every time a successful match is made.

Fig. 7 gives the description of PAR after this match.

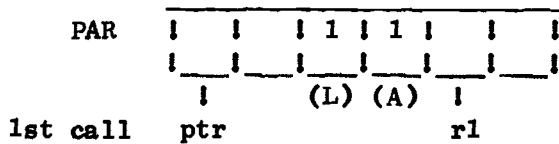


Fig. 7

Now Anal is called for (CONDITION), which is handled in the same way. Fig. 8 describes PAR after the successful recognition of (CONDITION).

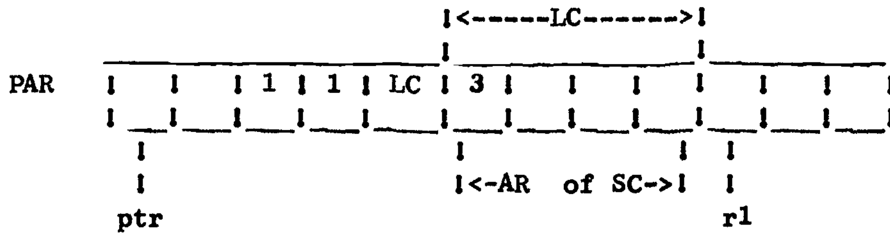


Fig. 8

The definition of (CONDITION) is

(SC) ´%and´ (ANDCL) | (SC) ´%or´ (ORCL) | (SC)

and ´i = 0´ is recognised as an (SC). Hence the alternative number is 3. LC is the length of the analysis record of (CONDITION).

Finally the (ELSECL) is recognised, and when this has been done, the whole alternative has been matched. Hence PAR(ptr) and PAR(ptr+1) are filled in, and the function gives the result 1 to Analyse.

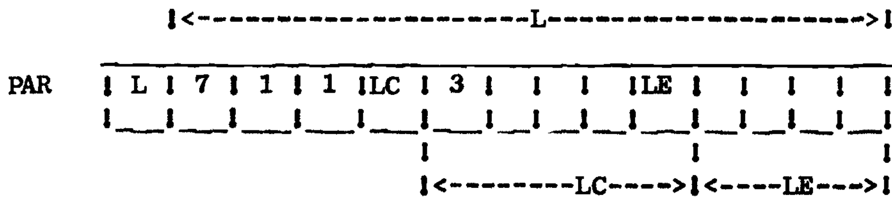


Fig. 9

Fig. 9 shows the final contents of PAR. The alternative we were considering is the 7th of (ACTIVEST).

7.7 Processing the Symbols * and ?.

So far we have made no reference to the negative numbers that occur in SYN, to indicate the possibilities of lists or empty expressions. Checks are made in two places.

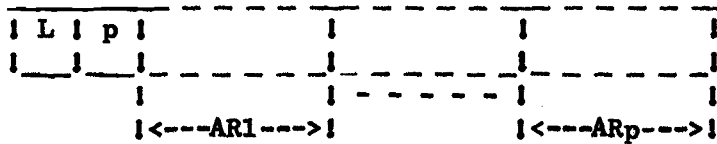
1) If a literal is found in an alternative, and is

followed by a -1, the negative number is skipped if the literal matched with the phrase being examined. If the literal does not occur in the phrase, however, 0 is entered in the array PAR and the processing continues with the same alternative.

2) Let us suppose that SYN(i) has the value j, and Anal has just been called for j. Then if SYN(i+1) is negative the following process takes place.

   a) If SYN(i+1)=-1, no action is taken if the result of Anal was 1. If it was zero, the value 0 is entered in PAR, and the process continues with j set to SYN(i+2).

   b) If SYN(i+1)=-2, and the result of Anal was 1, we must consider the possibility of a list. Hence if AR(p)=',' then m is set to p, and Anal is called again for j. This process is repeated until either AR(p)#',' or the result of Anal is 0. Space has been reserved in PAR for the number of occurrences of the phrase, and this number is inserted before the analysis record of the first occurrence. Fig. 10 shows the analysis set up by a list of p instances of the phrase name N.



where L = length of analysis record

p = number of N's

AR1 ... ARp are the analysis records of the N's

Fig. 10

c) If SYN(i+1)=-3, then the value 0 is entered in PAR when the result of Anal is zero. Otherwise we proceed as for (b).

7.8 Built in Phrases

The syntax given in Appendix A is not complete. Eight phrase names are not defined, because an ad hoc method of analysing them was thought better than a rigorous syntax description and analysis. For these eight, LINE(n)=0, (n=1,...8). A jump to the appropriate switch in Anal gives the method for dealing with these phrase names.

1) NAME

Names are read into the dictionary by Read or print, and replaced by the special sign '%', followed by a number which gives the position of the name in the dictionary. Hence to analyse a name, Anal merely checks for the '%', and copies the number following it into the analysis record.

Example

(OPD) = '('(SELECTOR)?(EXPR)')' ! (CONST) !

(NAME)(PLIST?) ! (LB)?

If 'p' is to be recognised as an operand, the analysis record is given by Fig 11.

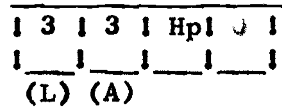


Fig. 11

where Hp is the hash code of p.

2) CONST

The internal form of a constant is described in detail later. Anal changes the string of digits, with the symbols '.' and '@' into the internal form. At this stage, the only constants recognised are integers and reals. (Rationals and long reals are created when an expression is evaluated.) The string of characters that represents the number is changed into that number. If it is real, it is stored in an integer location by the IMP statement

I = INTEGER(X).

Then the number, preceded by a code number indicating its type is copied into five bytes of PAR.

Thus the analysis record for '15' as an operand is given by Fig. 12.

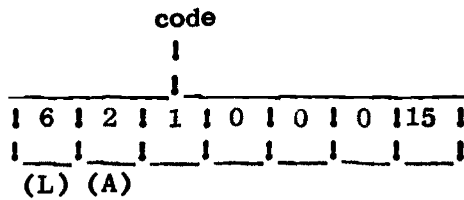


Fig. 12

3) EXPR

The usual syntactic definition of an expression as

(EXPR) = (OPD)(OP)(EXPR) ¦ (OPD)

is highly unsatisfactory when evaluating expressions, as the structure obtained is completely unlike that which the expression implies arithmetically. It was therefore decided to maintain a 'flat' structure in the analysis of expressions. The method of analysing an expression can be described by a short flow diagram, Fig. 13.

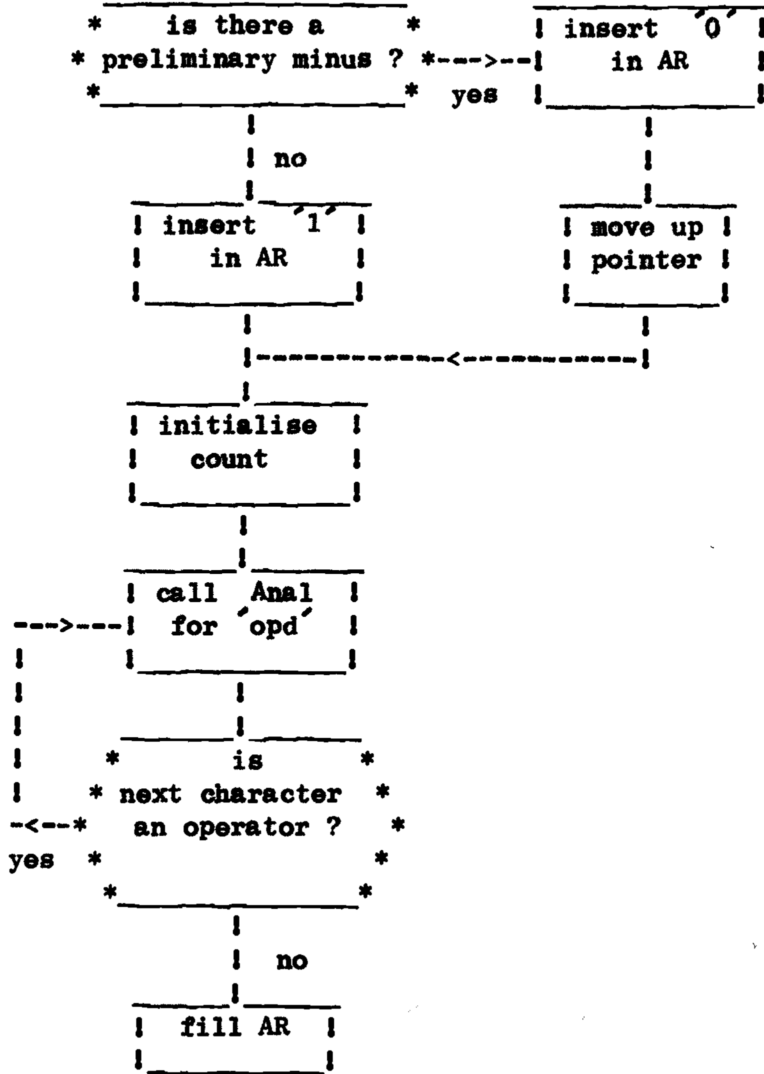


Fig. 13.

The analysis record for the expression 'a*b' is given by Fig. 14.

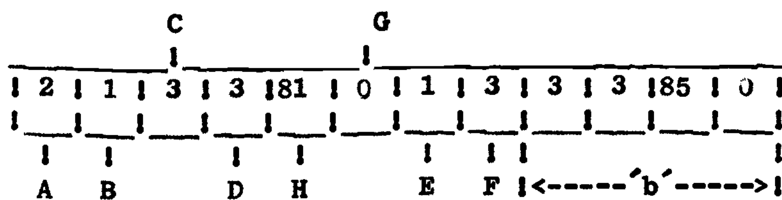


Fig. 14

where

A gives the number of operands

B is 1 if the expression is preceded by a plus (possibly implied) and is 0 if the expression is preceded by a minus.

C is the length of the first operand

D is the alternative of the first operand

E is the length of the operator

F is the alternative of the operator

G is the length of the list following the name

H is the hash code number of ´a´.

4) PATTEXPR

This would be described syntactically as

(PATTEXPR) = (PATT)(OP)(PATTEXPR)!(PATT)

Hence the same arguments are used as against a rigorous syntax of (EXPR), and the method of analysing this phrase name is exactly that described in (3).

5) PRIMES

It is far simpler to count the number of primes occurring in a (PLIST) than to define (PRIMES) rigorously. Primes are used to denote differentiation, as in f´(x).

The analysis record of this as a (BASIC OPD) is given by Fig. 15.

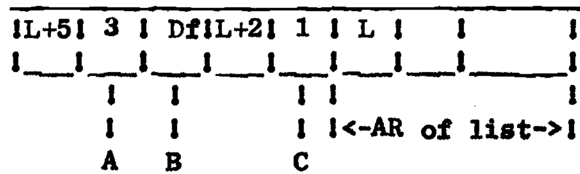


Fig. 15

where

A is the alternative of the operand

B is the position of f in the dictionary

C gives the number of primes

6) LB

This is another case similar to (3). The rigorous definition of (LB) would be

(LB) = (´:´)?(LEXPR) ´:´ (LB) ! (´:´)?(LEXPR).

The analysis record obtained for ´: 5 : 7´ is given by Fig. 16.

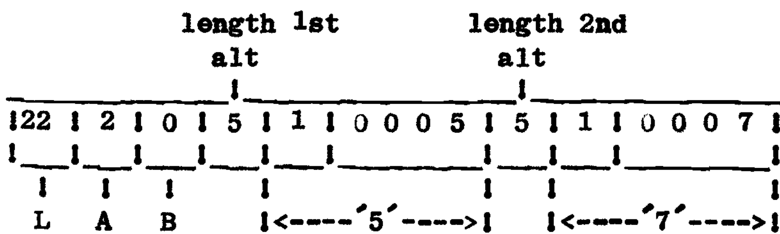


Fig. 16

where

L is the length of LB

A gives the number of LEXPRs in LB

B is set to 1 for a full label, and 0 for an abbreviated label

7) Lists

By either method of analysing syntax, lists can cause an unnecessary amount of back tracking. For instance, consider the list

(a,(b+c),(d+(e+f)),g)

Once a left hand bracket has been recognised, the first right hand bracket encountered is assumed to be its partner. Therefore

a,(b+c

would be tested as members of a list. This would fail, since ´(b+c´ would be faulted. Next

a,(b+c),(d+(e+f)

would be tested, and this would also fail. It can be seen that a great deal of time will be wasted by this method. To avoid this, AML counts the brackets it encounters for the built-in phrase (LIST), and only when the correct matching bracket is found will the elements of the list be processed. The algorithm for counting brackets is:

1) Count = 0.

2) Read character.

3) If it is ´(´, add 1 to count.

4) If it is ´)´, subtract 1 from count. Result is zero if count is negative. Required bracket has been found if count=0.

5) Go to (1) unless text is ended.

6) Result is zero, unless count is zero.

8) (TEXT)

The characters between quotes are stored in ALG. Hence the analysis record is merely a record of the position at which the text is held. (See Chapter 9 for details). Double quotes inside the text that are intended to represent a single quote are also copied into ALG; the alteration to the text is done when printing occurs.

VIII Implementing AML - Organisation of Store

The steps given for executing an unlabelled statement were two-fold:

a) Find the analysis record of the statement

b) Obey it.

Part (b) is dealt with by a routine called Exec, which assumes that it is examining the analysis record of an (ACTIVEST). Hence the first thing it does is to discover which alternative of the (ACTIVEST) it is processing, and branch to the piece of program which deals with that alternative.

The alternatives of an (ACTIVEST) can be grouped as follows:

1) '%local' and '%empty' statements.

2) Conditional instructions.

3) Array and routine declarations.

4) (UNCONDST).

The conditional instructions take three forms:

(UNCONDST) '%if' (CONDITION)

(UNCONDST) '%unless' (CONDITION)

(IU)(CONDITION) '%then' (ELSECL)

For each of these, the condition is tested, and if the required condition is returned (i.e. TRUE for '%if' and FALSE for '%unless'), the (UNCONDST) is obeyed.

(UNCONDST) is also dealt with by Exec, again making a jump to the appropriate piece of program. Hence a rather

sketchy algorithm can be given for the routine Exec as described so far.

1) Jump to the piece of program dealing with the alternative being processed.

....

2) Conditional statements - work out the condition. If the required one is given, go to (4).

3) Return if the (UNCONDST) is not to be executed.

....

4) Uncondst - jump to the piece of program dealing with the current alternative of (UNCONDST).

The (UNCONDST) which will be discussed first is the assignment statement, in its simplest form, i.e.

(NAME) ´=´ (EXPR)

which assigns an expression to a scalar variable.

Since there are no declarations for scalar variables, this may be the first time that the NAME in question has been mentioned. Also the (EXPR) being assigned to it may be of any type. There are three things to be considered here.

1) What information is there already available?

2) What information must be stored?

3) How can this be done?

1) When the statement was first read in, the NAME was placed in a dictionary, and was replaced in the statement by a unique number, which is the hash code of the name. This number is used to represent the name throughout the implementing program, and

at this stage is all the information available.

2) The information that must be stored is of course, the value of the expression. Since no information about the type is attached to the name, a type must also be given to this value. The possibilities are:

   integer

   rational

   real

   long real

   algebraic

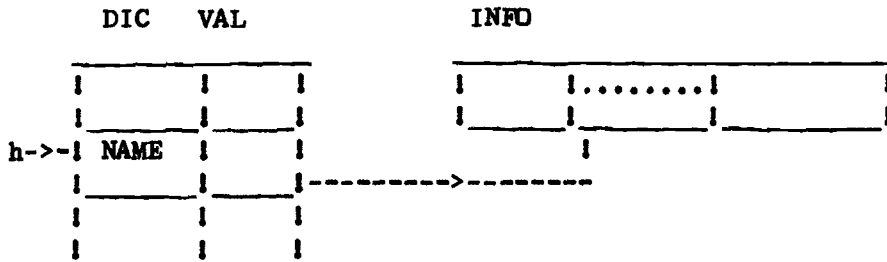
   The discussion has assumed that the name has not been previously assigned a value. If this is not the case, we have two possibilities

   a) The old value should be overwritten.

   b) The name has been declared '%local', so the old value must be saved before the new value can be stored.

3) Bearing these considerations in mind, we can discuss the way in which the information is to be stored. Since each name is associated with a unique number between 0 and 255, an array called VAL whose bounds are also 0-255 is used. Hence if h is the hash code of the NAME, VAL(h) will give information about the contents of NAME. However the amount of information to be stored depends on the kind of value that NAME has, and so it was decided to use another array, INFO, to hold the information, while

VAL points to the place in INFO where it is held. Fig. 1 shows this situation.



h is the position of NAME in DIC

VAL(h) points to the position in INFO where the information is held.

Fig. 1

8.1 How INFO is Organised.

INFO was chosen to be a short integer array; an integer array results in a fair amount of waste space, while a byte integer array would involve an unnecessary amount of shifting to get values in and out. It is in fact used to describe all names; arrays, routines and parameters, as well as scalar variables. Its bounds are 1 to 5000. The variable iptr is used to point to the next available location of INFO. Whenever there is not enough space left after iptr to store the information describing the NAME being considered, a garbage collection routine is called. This will be described in Chapter 9. It is possible that a block of information in INFO may no longer be required. This happens for three reasons.

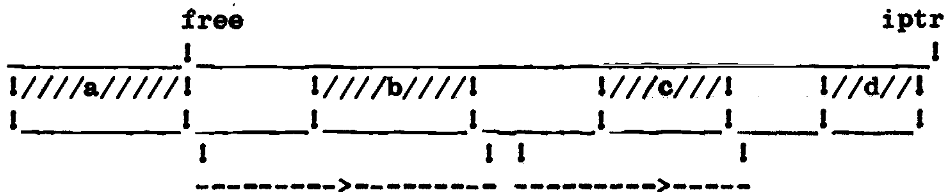
1) The information describes a value type parameter to a routine that is being ended.

2) The information describes a variable or array that is local to a routine that is ended.

3) The instruction ´%empty´ requires the information to be removed.

The blocks of information that are no longer needed are chained together, with a variable called free pointing to the beginning of the first block. The garbage collection routine, when it is called, moves these free blocks to the end of INFO, thus making them available for re-use. Fig. 2 shows the alteration.

Before



After

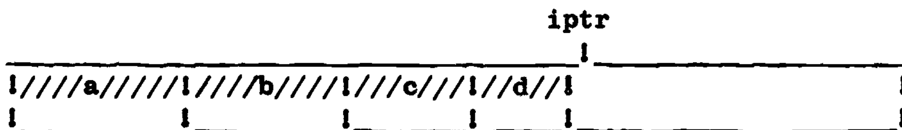


Fig. 2.

8.2 Scalar Variables

Now let us examine the kind of information that must be stored in INFO. For a scalar variable, the values that may be stored are

integer - takes up one machine word

real	- takes up one machine word
rational	- takes up two machine words
long real	- takes up two machine words
algebraic	- takes up an unknown number of bytes.

It was decided that five locations would be reserved for a scalar, no matter what type of expression was assigned to it. This means that space is wasted in some cases. However this method was thought preferable to that of assigning the exact amount required in a particular case, and so possibly having to re-allot space later,

The five locations are organised as follows:

1) integer

```
 _____________________________
! 1 !   !   !   !   !   !
!___!___!___!___!___!___!
  !   !-----! !-----!
code  value    empty
```

2) real

```
 _____________________________
! 2 !   !   !   !   !   !
!___!___!___!___!___!___!
  !   !-----! !-----!
code  value    empty
      (put into an integer
      location using the mapping
      function 'integer')
```

3) rational

```
 _____________________________
! 3 !   !   !   !   !   !
!___!___!___!___!___!___!
  !   !-----! !-----!
code  numer-  denom-
       ator   inator
```

```
4)  long real          ________________________________
                       | 4 |    |    |    |    |    |
                       |___|____|____|____|____|____|
                         |   |------------------|
                       code  value (put into 2 integer
                             locations using the mapping
                             function 'integer')

5)  algebraic          ________________________________
                       | 5 |    |    |    |    |    |
                       |___|____|____|____|____|____|
                         |   |         |---------|
                       code  |            empty
                             pointer to access the
                             algebraic expression
                             which is stored in ALG
```

It will be seen that ALG is used in another context; it holds the algebraic expressions that form the value of some variable.

The method used for assigning a value to a variable is given here. This is correct only for variables that are global in the program.

> Let H be the hash code of the name. Then if VAL(H) is zero, no value has previously given to this name. Therefore the next five locations of INFO after iptr are used. VAL(H) is set to point to the first of these. If there are not five locations available, the garbage collection routine must be called first. If VAL(H) is non-zero, the value being assigned will overwrite the current value. If the old value is an algebraic expression, it must be erased from ALG by replacing the first entry by 255 (See Chapter 9). The new value is put into INFO according to the method described above.

If the new value is algebraic, it must be copied into ALG, and pointers set to the beginning and end of the expression. Again, see Chapter 9 for a full description.

The algorithm for extracting the value of a variable is very similar, and so will be given here.

1) Let H is the hash code of the name.

2) If VAL(H) is zero, no value has been assigned to this name, so it is an algebraic constant.

3) The location of INFO that VAL points to gives the type of the value assigned to the name. The steps taken next depend on this number.

4) If it is 1, the value is an integer, and its value is given by the equation

```
j=INFO(k+1)<<16!INFO(k+2)
```

where k=VAL(H)

5) If it is 2, the value is a rational, whose numerator is given by

```
INFO(k+1)<<16!INFO(k+2)
```

and denominator by

```
INFO(k+3)<<16!INFO(k+4)
```

6) If it is 3, the value is real, and is given by

```
i=INFO(k+1)<<16!INFO(k+2)
x=real(i)
```

7) If it is 4, the value is a long real and is obtained in a manner similar to that of (6).

8) If it is 5, the value is algebraic. Let

p=INFO(k+1). This will give the position of the expression in ALG. (See Chapter 9.)

8.3 Arrays

Now let us consider the assigning of a value to an array element. Any expression may be assigned to an array element, so five locations of INFO are required for each element. However there is not a pointer in VAL available for each element, so the locations are saved when the array is declared, and the routine handling the assignment must discover which of these locations it is concerned with.

Suppose that an array A is declared with bounds L and U, i.e.

```
%array A(L:U)
```

and that an element A(i) is to have a value stored in it. Then the five locations that hold the value of A(i) begin at

```
INFO(A(L)+(i-L)*5).
```

The value of i must be calculated at the time of the assignment, but the value of L and the position of A(L) are determined by the declaration. Moreover if two arrays are declared in one statement, e.g.

```
%array A,B(L:U),
```

the value of L is common to both, although the positions of A(L) and B(L) are of course different.

For a one-dimensional array only the value L is needed. However for multi-dimensional arrays the situation is more complicated. Let us consider the two-dimensional array

%array C(1:5,1:4)

For this, 20 sets of five locations must be saved on INFO. This is essentially storing a two-dimensional array as a single dimensioned array. Let us consider the more general case, where C is declared as

%array C(L1:U1,L2:U2)

and the element C(i,j) is required. The first position on INFO of C(i,j) is given by

C(L1,L2)+((i-L1)*(U2-L2+1)+j-L2)*5

In this case, i and j must be calculated at the time of the assignment, but the values L1, L2 and U2 are known when the declaration is made. Moreover these values are required for all arrays in the statement declared to have the same bounds. Hence they can be stored in one place, and accessed by all the arrays concerned. To calculate the position of C(i,j), only L1, L2 and U2 are needed. However, to check that i and j lie within the bounds of C, U1 must be recorded as well. Therefore for an n-dimensional array the bounds be stored are

L1, U1, L2, U2, ... Ln, Un.

These values are said to form the Dope Vector of the array, and are also stored in INFO. It is necessary to have a code number for an array, to distinguish it from a scalar. This code number is chosen to be 6. Hence when an array is declared, the space saved for it on INFO is as given in Fig. 3.

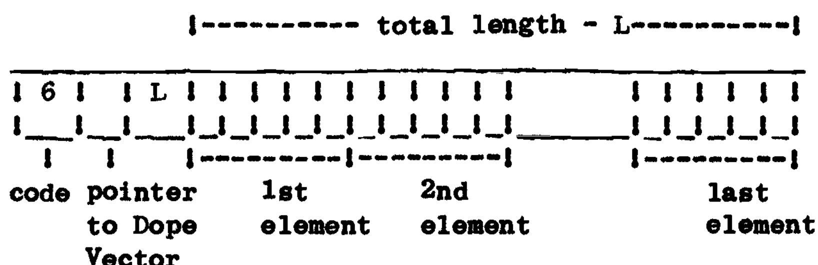


Fig. 3

The length L is used when removing an array description from INFO.

The Dope Vector for a two-dimensional array would be

N is the number of arrays using this dope Vector. It is decreases by one every time an array description is removed, and when the last array description has been deleted, the Dope Vector is also removed. The second entry is the number of dimensions of the array.

The algorithm following is for the general form of an array declaration, excluding the word ´%local´. In other words, for

%array (ARDEFN)*

where (ARDEFN) = (NAME)*´(´(BOUND PR)*´)´.

1) Consider the first bound pair of (ARDEFN).

2) The Dope Vector is constructed, as described in (3) to (8).

3) Let N be the number of names preceding the list of bounds, and D the number of dimensions. 2*D+2

spaces are saved on INFO, Dv being the first of these. N and D are stored in the first two locations. Let L=1.

4) Consider the first set of bounds.

5) The lower and upper bounds, J and K, are evaluated, checking that J<K. Set L=L*(K-J+1).

6) J and K are stored in the next two locations of INFO.

7) (5) and (6) are repeated if there is another set of bounds.

Steps (8) to (11) process the names of the arrays.

8) Consider the first name of (ARDEFN).

9) Save 5*L+3 locations on INFO.

10) If H is the hash code of the name, then VAL(H) is set pointing to the first of these locations.

11) The code number, 6, is stored in the first location, the position, Dv, of the Dope Vector in the second, and the value 5*L in the third. The remaining locations will be used to hold the values of the individual elements.

12) Steps (9) - (11) are repeated if there are any more names.

13) Go to (2) if there are any more (ARDEFN)s.

Now let us consider the assignment of a value to an array element. This process consists of two parts.

a) Find the position in INFO which is to be used for the element in question.

b) Fill these five locations with the information which describes the value.

The second part of this process is identical to that used for scalar variables, and so will not be described here.

If an array A has n dimensions, given by

L1:U1, L2:U2, ... Ln:Un,

then the position of an element A(I1,I2,...In) is given by the formula

k + 3 + Sn*5

where k is the first position on INFO of the description of A, and Sn is given by

S1 = I1 - L1

Sj = Sj-1*(Uj - Lj +1) + Ij - Lj, j=2,3,...n

The steps taken in AML to find this position are

1) Let m point to the description of A.

2) Let d point to the beginning of the Dope Vector (INFO(m+1))

3) Check that the number of dimensions coincide.

4) Set k=0 and s=0. Do (5) - (8) for j=1,1,n

5) Calculate the value I if the next subscript.

6) Check that it lies between Lj and Uj.

7) If j=1, set t=1, otherwise t = Uj-Lj+1

8) Set s = s*t + I - Lj, k = k+s.

9) The required position is m+k*5+3, which points to the first of the five bytes reserved for the element.

The algorithm for finding the position of an array element to read its value is, of course exactly the same. Two points must be made. First, if the subscripts of the array do not all evaluate to integers, the array element is treated as an algebraic constant. Secondly, even if the subscripts do have integer values, there may be nothing stored in the element. In this case there is a zero in the position of INFO that VAL points to. The element is treated as an algebraic constant.

8.4 Routines and Functions

The declarations of routines and functions are treated in exactly the same way, except that the code numbers are different. For a routine the code number is 8, and for a function it is 7. Therefore discussion will be confined to routines, and the first to be considered will be routines without parameters, for example

%routine alpha %at 1:1.

The information that must be stored for this routine is

1) the code number.

2) the position in the Storage Tree at which the body of the routine is held.

3) the number of parameters, in this case, zero.

Hence three locations are reserved on INFO, and VAL(H) is set pointing to the first of these (where H is the hash code of alpha). The code number, 8, is stored in the first of these locations. The position, S, of the label in

the Storage Tree is calculated, and put in the second location, and zero is put in the third location. Fig. 4 describes this situation.

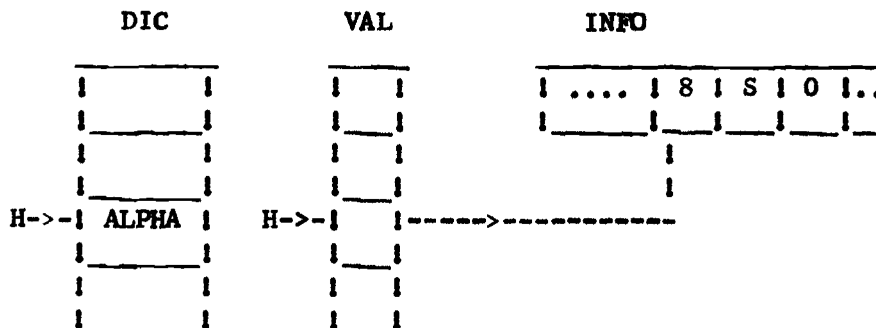


Fig. 4.

When the routine is called, since there are no parameters, the block at S is executed immediately. When this has been done, control is passed to the statement to be executed after the routine call.

8.4.1 Parameters.

If the routine has parameters, more space is required to store information about them. At this stage two items of information are available:

1) the name of the formal parameter

2) its type.

Therefore two extra locations are saved for each parameter. The first of these contains the code number for the type, and the second stores the hash code of the name. The code numbers for formal parameters are given below.

9	%value	parameter
10	%name	parameter
11	%array	parameter

12 %fn parameter

13 %routine parameter

14 %label parameter

If the routine beta is declared as

%routine beta(%value n,%name r) %at 2:1,

then Fig. 5 describes the configuration that is set up, where T is the position of 2:1 in the Storage Tree.

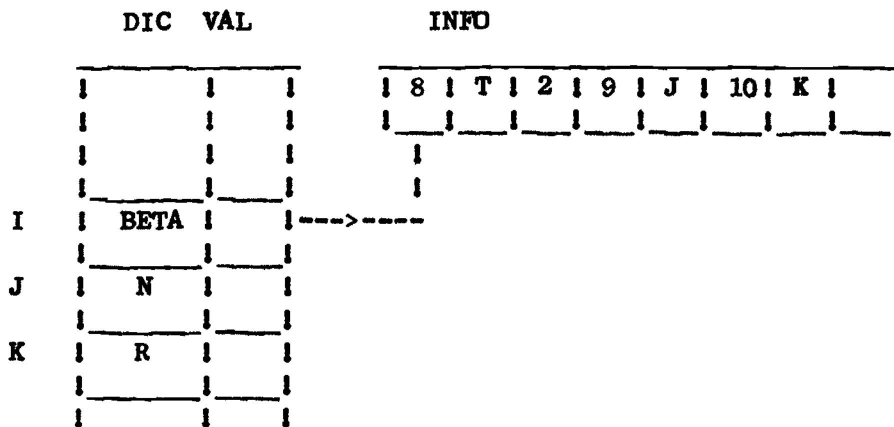


Fig. 5

8.4.2 Calling a Routine - Value Type Parameters.

Let us consider a routine that has one value type parameter. The routine is declared as

%routine gamma(%value v) %at 3:1

and the call could be

gamma(3)

Before the body of the routine is obeyed, the interpreter must

a) Save any value attached to v.

b) Create a new location, also called v.

c) Evaluate the actual parameter, and put that value,

in this case 3, in the new v.

d) When the routine is finished, the interpreter must restore the old value of v.

NOTE:

The AML implementation of value type parameters follows that of IMP. The parameter can be regarded as an initialised local variable; the programmer is at liberty to change its contents inside the routine. Unless he does, the value which is passed in at the time of the routine call is used every time the formal parameter is named.

Evidently a push-down stack must be used. The routines may be used recursively, and it is possible for several values of v to be already stored when this call is made. This is true whether gamma is called from itself or from another routine. Initially, however, the discussion will be concerned with a routine called from the main program. The push-down stack consists of cells taken from from a free list whose two components are called CAR and CDR. This list is constructed by declaring CAR and CDR as arrays. The first free cell is given by a pointer called psl.

The method used for AML for implementing routines differs from the standard method of using a stack and Dijkstra Display, that is associated with block structured languages like IMP and ALGOL. It must be remembered that blocks in AML are not the same as the blocks in the above mentioned languages. The difference can be shown in a

diagram, as seen in Figs. 6 and 7.

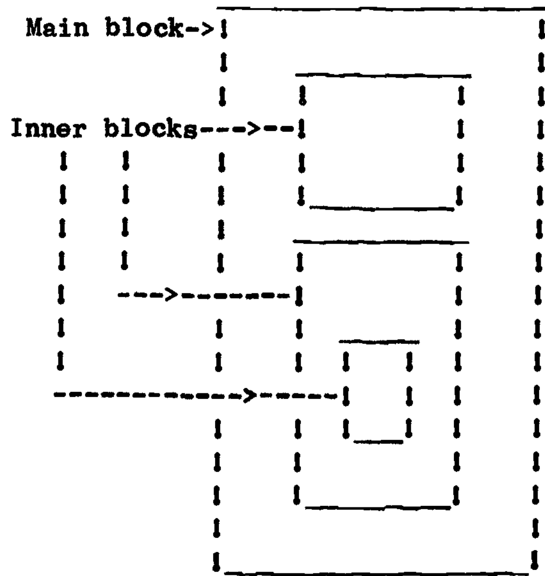


Fig. 6

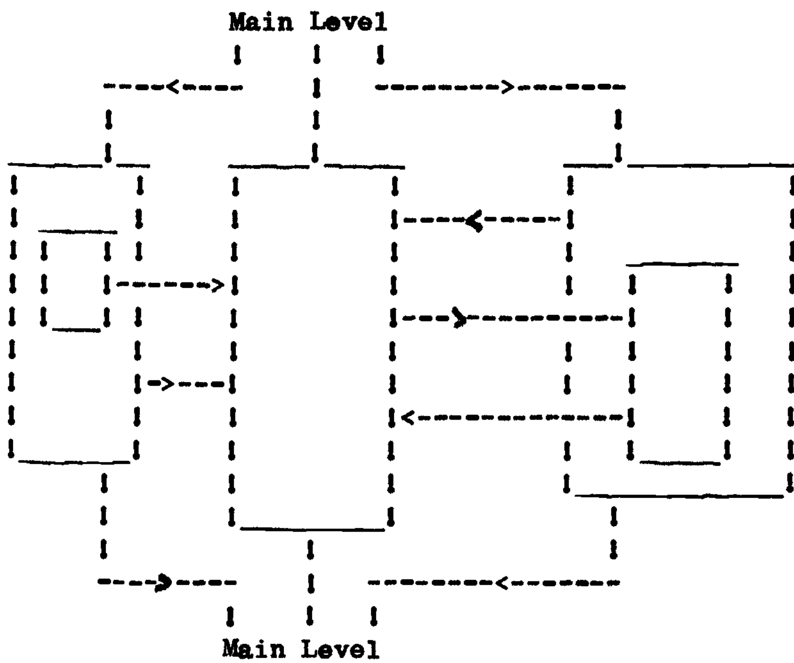


Fig. 7

The arrows show some of the possible paths of control.

Hence there is no textual level which can be used as the basis of stack organisation, as in IMP.

A cell is removed from the free list by the statements

```
k=psl ; psl=psl+1
```

Then the cell k is available for use. To set up an actual %value parameter when a routine is called, a cell k is removed from the free list. If I is the hash code of the formal parameter, the contents of VAL(I) are copied into CDR(k). Then VAL(I) is set to point to K. Five new locations are reserved on INFO, and CAR(K) is set pointing to the first of these. Then the actual parameter, is evaluated, the result being put in the new locations of INFO.

If T is the position of 3:1 in the Storage Tree, Fig. 8 shows the situation after the statements

```
%routine gamma(%value v) %at 3:1
...............
v=5
gamma(3)
```

have been executed.

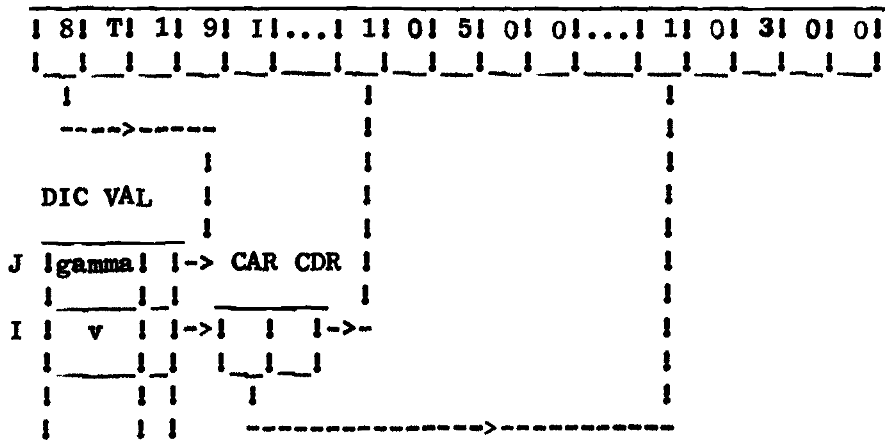


Fig. 8

INFO has bounds from 1 to 5000, and so CAR and CDR are numbered upwards from 5001. This ensures that it is easy to distinguish between a pointer to INFO and a pointer to a CAR-CDR cell when VAL is examined.

At the end of a routine, the value in CDR is replaced in VAL(I), the cells in INFO that CAR points to are emptied, and the CAR-CDR cell is returned to the free list. (See Chapter 9 for details.)

This method of setting up %value type parameters means that the description of the handling of assignment statements, given earlier, must be modified. The statement

NAME = EXPR

may possibly occur inside a routine that has a formal %value parameter called NAME. In this case, VAL does not point directly to INFO. So the method of finding the position in INFO in which the value of EXPR is placed, must be changed.

1) Set K=VAL(I).

2) If K is zero, then get five new locations, as before.

3) If K<5001, the K points directly to INFO.

4) If K>5000, K points to a CAR-CDR cell. Set J=CDR(K).

5) If J>5000, J also points to a CAR-CDR cell. Therefore set K=J and go to (4).

6) Otherwise, K=CAR(K) points to INFO.

As an illustration of a more complicated situation, let us consider this piece of program

```
    %routine gamma(%value v) %at 3:1
    v=5
3:1 ............
:1:5 gamma(7)
    ...............
    gamma(3)
```

This illustrates a recursive call on gamma. Gamma(3) is called from the main level of the program, and gamma(7) is called from inside the routine gamma, which is found at block 3:1. Fig. 8 describes the situation when the call 'gamma(3)' is made. Fig. 9 describes the situation when the statement labelled 3:1:5 is executed.

INFO

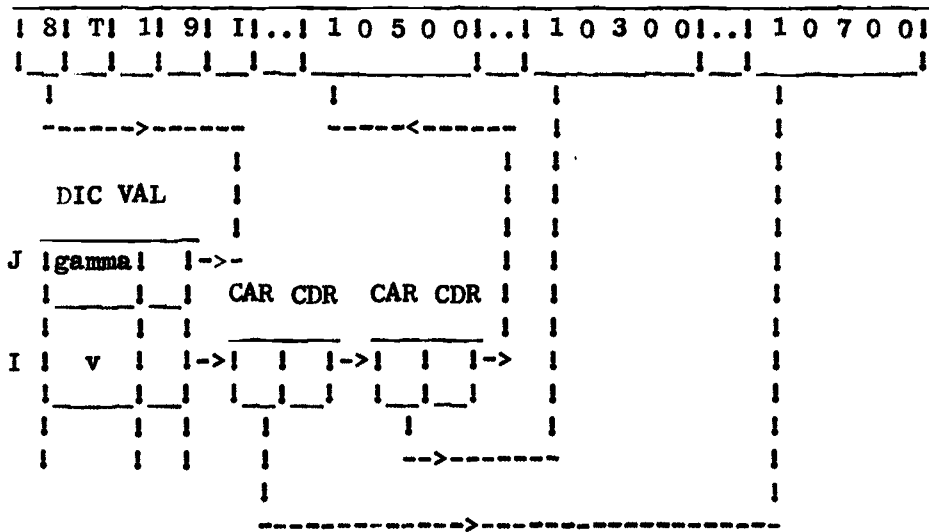


Fig. 9

8.4.3 %Name Type Parameters

The emphasis is different for a %name parameter. It is not the value of the actual parameter that is required, but the position of it. No extra locations are saved on INFO; instead the CAR of the push-down cell is set pointing to the description of the actual parameter, which must, of course, be a name. If the actual parameter has not been mentioned before (i.e. it is an algebraic constant at this point), then space is saved for it on INFO, but this is not lost when the routine is ended. Let us consider an example.

```
%routine delta(%name n) %at 4:1
n=5 ; g=1
delta(g)
```

When delta is called, the interpreter must do two things.

1) Save the old value of n.

2) Set the new n pointing to the locations of g.

As before, a cell, k, is extracted from the free list, and the old contents of VAL(j) are stored in CDR(k). (If j is the hash code of the formal parameter n.) CAR(k) is set pointing to INFO, to the position describing the actual parameter g. Fig. 10 shows this situation.

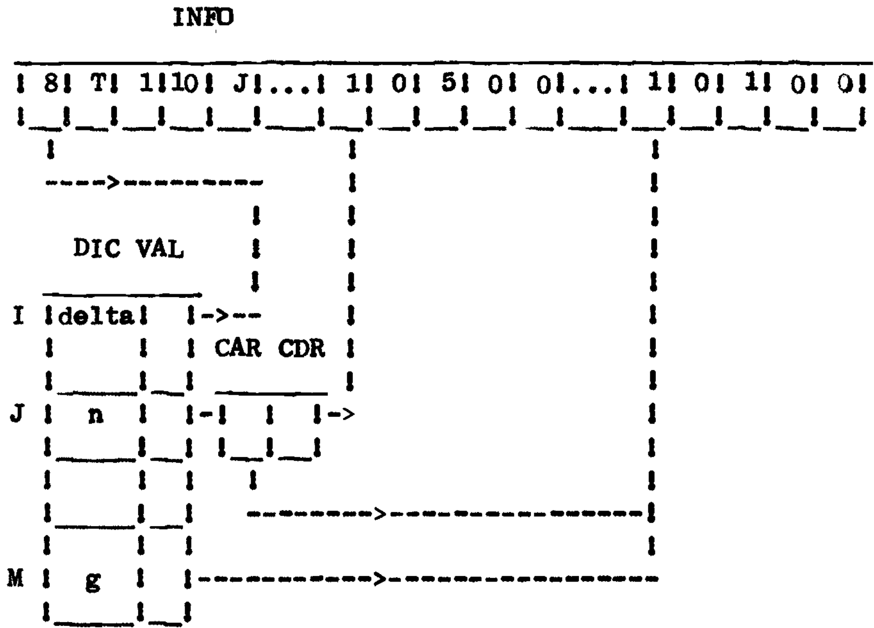


Fig. 10

Then if an instruction inside the routine refers to the formal parameter n, the operation required will take place on the actual parameter g. For instance,

4:1:4 n=2

will result in the value 2 being stored in g. When the routine is ended, the CAR-CDR cell is removed, but the locations of INFO must not be removed. To indicate this, the value in CAR for a name type parameter is stored as a negative number, the modulus of which points to INFO. Then the routine which removes the cells can tell that the

locations of INFO are not to be emptied.

8.4.4 Array Parameters

Since an array parameter refers to a number of locations, it acts like a %name parameter, and in fact as regards the setting up of the parameter description, they are treated exactly alike. An example which illustrates the call of an array parameter is

```
    %routine theta(%array a) %at 5:1
    %array b(1:10)
    ..........
5:1 a(1) = 5
    theta(b)
```

Fig. 11 shows the situation after 5:1 has been executed. I points to the position of 5:1 in the Storage Tree.

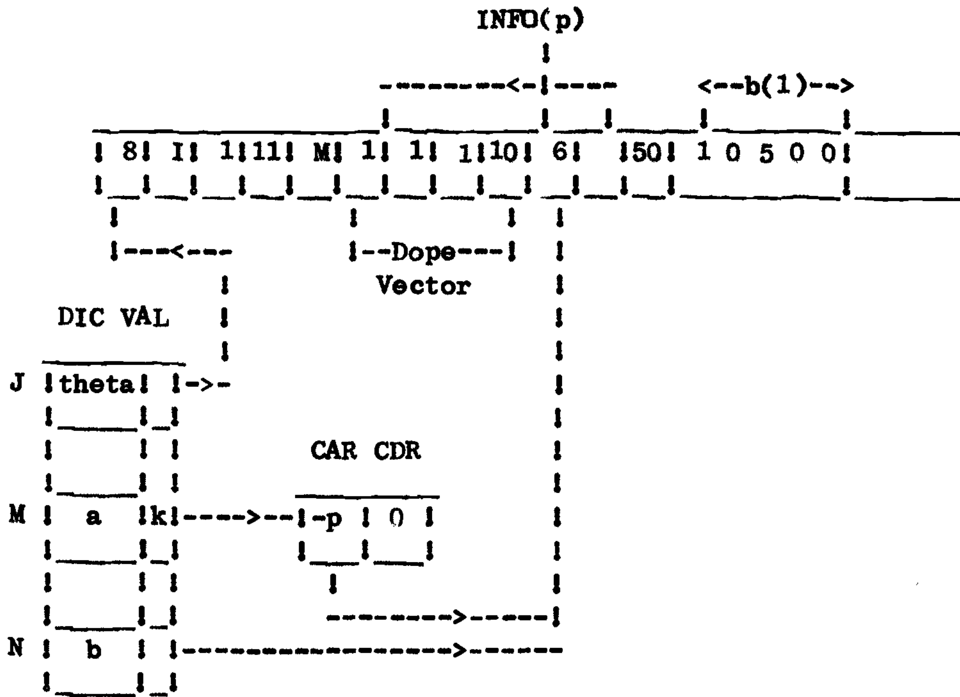


Fig. 11

The checking for dimensions and array bounds is done by the statements of theta, as described for statements at the Main level.

8.4.5 Routine and fn Parameters

These are treated in the same way as %name parameters; a negative number is placed in CAR, the modulus of which points to the routine or function description.

8.4.6 Label Parameters

This is a special kind of value parameter. The position in the Storage Tree given by the actual parameter is found. Two locations are saved on INFO. The first contains the code number 14, and the second, the position in the Storage Tree

that has been calculated. CAR points to these locations. Since CAR is positive, these locations will be removed when the routine is left.

8.4.7 %Local Variables

Local variables declared inside a routine must look like %value parameters, except that there is no value assigned to them initially. The declaration is, for example,

```
%local a,b,c
```

and the algorithm for making variables local is as follows:

1) Fault the program if it is not executing a routine.
2) Consider the first name.
3) Take a cell, k, off the free list.
4) Copy the value of VAL into CDR(k) and put the value of k into VAL.
5) Go back to (3) if there are any more names.

No locations are reserved on INFO at this stage, since the local value of a variable may be an algebraic constant (and initially all the names are). Thus the value in CAR for a local variable is used in the same way as the value in VAL is for a global variable. Fig. 12 shows the situation after the statements

```
    a=5
    %routine alpha %at 1:1
1:1 %local a
    ...............
```

alpha

have been executed.

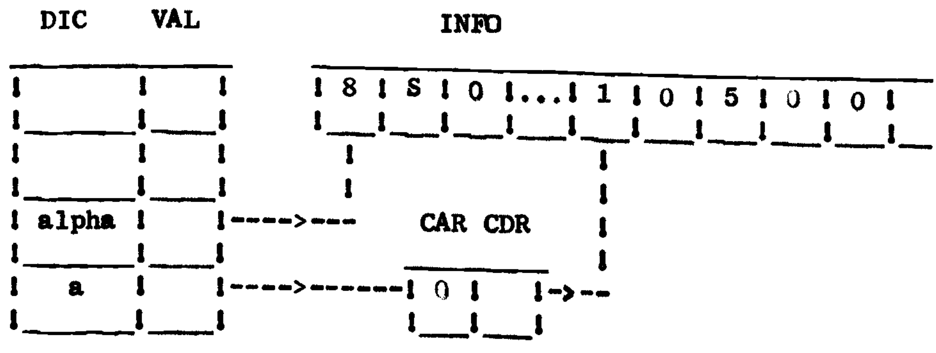


Fig. 12

IX Implementing AML - Garbage Collection

This chapter deals with all tidying up operations that are used in AML, and also describes how ALG is organised. It has three sections.

1) Chaining free locations.

2) Garbage collection for INFO.

3) Organisation of ALG.

9.1 Chaining Free Locations.

This must be done on three occasions: when an '%empty' declaration is obeyed, at the end of a routine, and when a fault occurs inside a routine.

a) The statement

%empty (NAME)*

sets the NAMEs in the list (they must all be global), back to the state of being algebraic constants. This is done by removing the pointer from VAL, freeing everything that hangs from it, and setting the contents of VAL to zero. A routine called Chainloc (see below) does this, so to execute the %empty declaration, Chainloc is called for each NAME of the list.

b) At the end of a routine. Routines may be called from each other up to a maximum of 15 levels. Hence when a routine is ended, it is important to release only those cells describing parameters of the routine that is ended, and not those declared earlier. Let us take an example.

```
        %routine beta(%value x,y,z) %at 2:1
        %routine gamma(%value v,w) %at 3:1
2:1:5   gamma(4,6)
        ...............
        beta(3,5,8)
```

Gamma(4,6) is called from within beta. Hence when the system leaves gamma, the CAR-CDR cells describing this call of gamma must be returned to the free list, but those which describe the parameters of beta are still required. Thus an array of pointers to the CAR-CDR stack is used. This array, called RTN, is used in the following manner. The current level of the dynamic calls of routines is held in a variable called level. RTN(level) points to the first of the CAR-CDR cells that is used for this call of a routine. Level is increased every time a routine is entered, and decreased when it is left. Fig 1. shows the connection between RTN and the CAR-CDR cells.

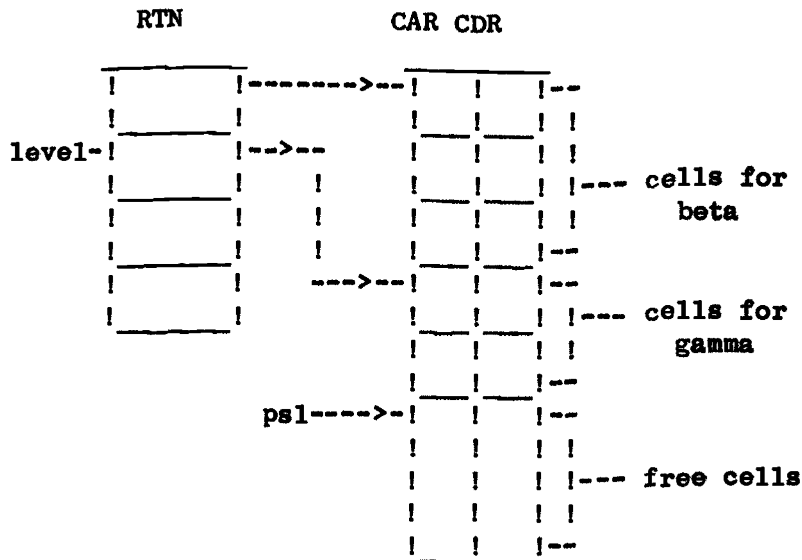


Fig. 1

Thus, when the routine is ended, Chainloc is called for each cell between RTN(level) and psl that points to a value type parameter. Then psl is reset to RTN(level), and level is decreased by 1.

(RTN is a byte integer array, so that in fact it is RTN(level)+5000 that gives the position on the CAR-CDR stack of the first cell used in the description for the parameters of the current routine.)

c) All faults in AML are trapped by calling the routine Flt, which does the following

1) Prints the error message.

2) Deliberately causes an overflow, which is trapped by an IMP %FAULT statement, thus returning control to the outer most level.

If a fault occurs inside a routine, this means that control does not go through the normal end of routine process. Hence in order to restore all variables to their

global state, a routine called Routine end is entered after a fault trap, while the level is greater than 0.

(Since the control word %WHILE is not available in IMP as implemented on the 4-75, the code to do this is

```
4:     %IF LEVEL>0 %THENSTART
       ROUTINE END; ->4
       %FINISH.)
```

d) Chainloc.

This routine sets up the free locations of INFO, ready for the garbage collection routine to consolidate them when it is called. Since any value may be stored on INFO, it is not possible to use a particular number to represent a free location of INFO. Instead, a pointer, free, is used and the free locations are chained together in the following manner.

At least two locations will be freed at one time; more often there will be a higher number. For each set of free locations, the first points to the next set of free locations, and the second gives the number of free locations in its set. The chaining is kept in order, i.e. no set may point to locations in front of itself. Free points to the first of the empty blocks, and the first location of the last block is set to zero. Initally free is set to 0, and if free is 0 when the garbage collection routine is called, there are no free cells available in INFO. Fig. 2 shows the situation when the locations marked A are about to be freed. The nearest block of locations in the chained list, that occurs in front of A, is marked j. This points to the next

set in the chain, at f, which is behind A. Therefore A is inserted in the chain between j and f.

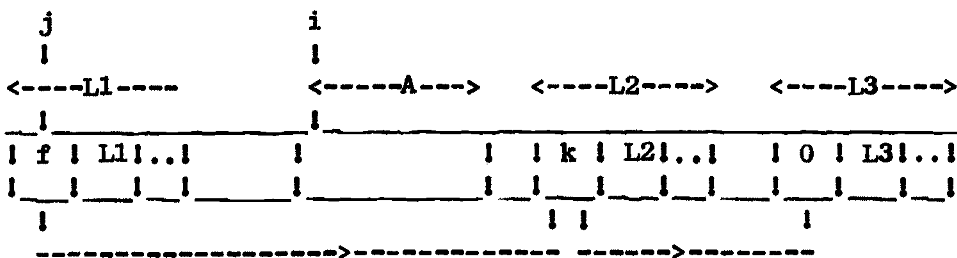


Fig. 2

When Chainloc has been executed, the pointers are changed, as in Fig. 3.

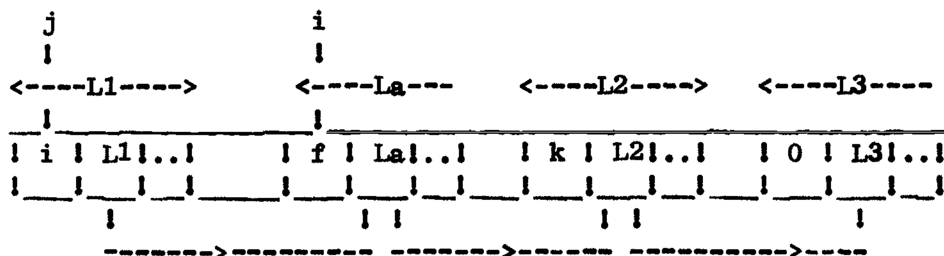


Fig. 3

Chainloc has two things to do. It discovers how many locations are to be freed, and then it adjusts the pointers in the manner described above. If there are any locations in ALG referred to by the locations being released, the first of these must be set to 255. Also if an array is removed and if it is the last to use a particular Dope Vector, then the locations of the Dope Vector are also emptied.

Let 'first' point to the beginning of the block of locations which are to be added to the chain of unwanted cells of INFO. Then INFO(first) will determine how many locations are in the block. The rules for finding this

number, which will be put in ´length´, are as follows.

1) If INFO(first) lies between 0 and 6, these cells were used to describe a scalar variable. Therefore length = 5. Moreover, if INFO(first) is 5, the locations of ALG accessed by this description must be released. (See later for a description of how this is done.)

2) If INFO(first) = 6, the cells were used to describe an array. Fig. 4 shows how an array and its Dope Vector are stored in INFO,

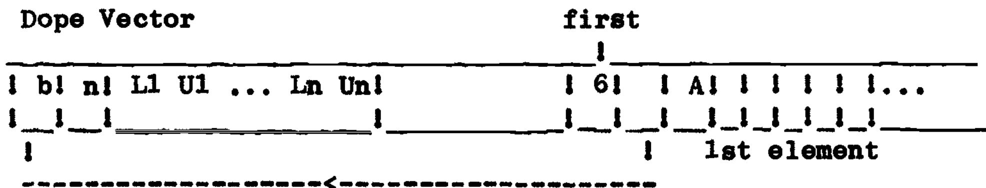


Fig. 4.

where b is the number of arrays using the Dope Vector.

n is the number of dimensions.

L1, U1,...Ln, Un are the bounds taken up of the array

and A is the length taken up by the elements of the array.

(See Chapter 8 for a detailed description.)

INFO(first) points to the Dope Vector of the array. If b is 1, then the Dope Vector must also be put in the chain of free cells. For this, it can be seen from the diagram that length = 2*n+2. If b is greater than 1, other arrays which are still in use will access the Dope Vector.

Therefore b is reduced by 1, since this array no longer uses it, but the Dope Vector is not removed.

For the array itself, length is set to A+3, since A gives the number of locations used by the elements of the array.

3) If INFO(first)=7 or INFO(first) = 8, a routine or function is to be removed. Fig. 5 shows the description in INFO of a routine.

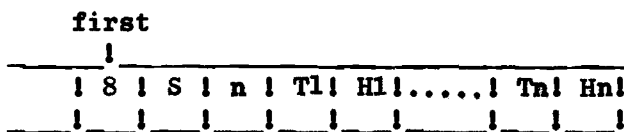


Fig. 5

where S is the position in the Storage Tree of the block of statements that define the routine.

n is the number of parameters.

Ti defines the type of the i-th parameter.

Hi gives the hash code of the i-th parameter.

Hence length = 2*n+3.

Once length has been set, the cells at 'first' can be added to the chained list. Chaining down the list from free, the last set of cells in front of first must be found. Let j point to these. Then INFO(j) points to the next set of cells in the list, which will be behind those of first. (See Fig. 3.) These are found at f. Thus the alterations to be made are

INFO(first) = INFO(j)

INFO(j) = first

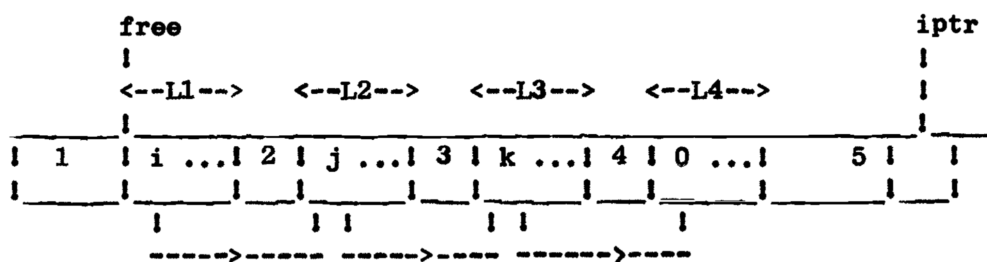
INFO(first+1) = length

Terminal cases, where first occurs before free, or after the last entry of the chained list must also be dealt with. The differences between these and the general case are too trivial to be discussed here.

9.2 Garbage Collection for INFO

INFO is a short integer array whose bounds are 1 to 5000. The variable iptr is used to point to the next available location of INFO. Whenever it is found that iptr is about to overflow the end of INFO, the garbage collection routine is called. This must do two things; first it must move up all the values that are still required, thus leaving all the empty locations at the end of INFO once more. Secondly it must alter all pointers to INFO. Fig. 6 shows INFO as it is just before the garbage collection routine is called, and also the situation after it has finished.

before



after

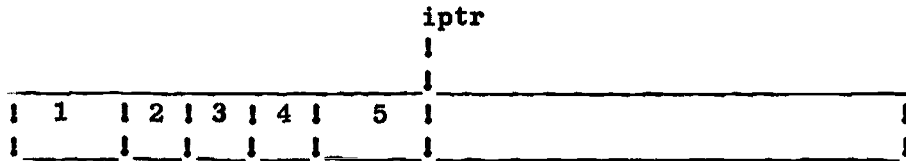


Fig. 6

It will be noticed that

a) The locations before free, marked 1, will not be moved

b) The locations marked 2 will be moved a distance L1

c) The locations marked 3 will be moved a distance L1+L2.

d) The process continues until finally the locations marked 5 will be moved a distance L1+L2+L3+L4.

Also

a) If there is a pointer to a locations in set 2, it must be decreased by the value L1.

b) Similarly, a pointer to a location in set 3 must be decreased by a value L1+L2. Etc.

The routine Garbage collect fills two local arrays, POSN and ADD as follows. POSN(p) contains the beginning of the p-th block of free cells. ADD(p) contains the total length that the locations before the p-th block are to be moved up. Hence, referring to Fig. 6, POSN(1)=free, ADD(1)=0, POSN(2)=i, and ADD(2)= L1. The locations are moved at the same time as the values of POSN and ADD are filled in. Here is the algorithm,

1) Set p=1, length=0, q=free.

2) Set POSN(p)=free, ADD(p)=length. Increase length by the value of INFO(free+1)

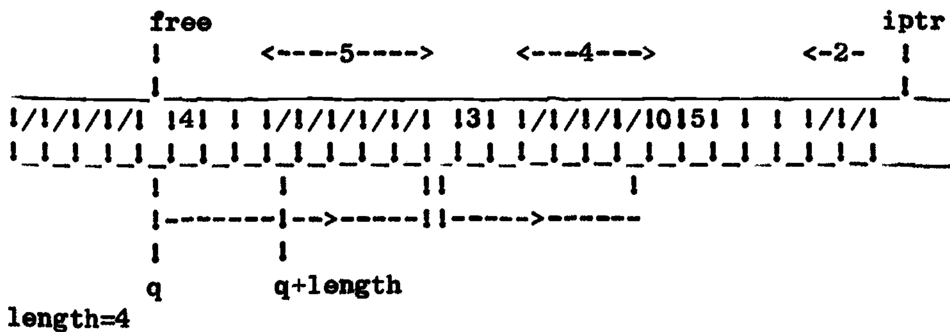
3) Increase p; set free=INFO(free) to get the next block of empty locations.

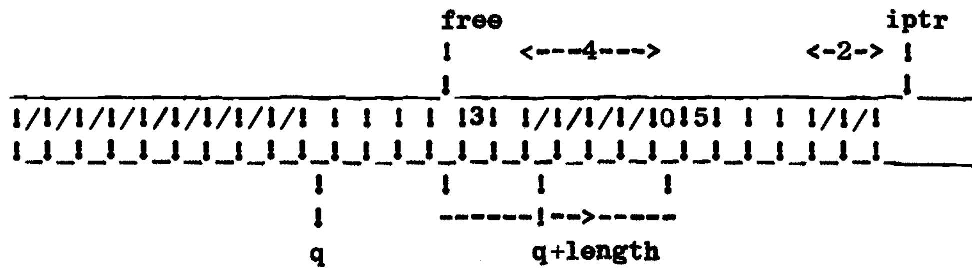
4) If free=0, the last empty block has been processed. Go to (6).

5) Q points to the next location of INFO that can be filled. The values that are to be moved are in q+length to free-1. Copy these into INFO, beginning at q. (Fig. 7 illustrates this.) Reset q to point to the location after the last value copied in. Go to (2).

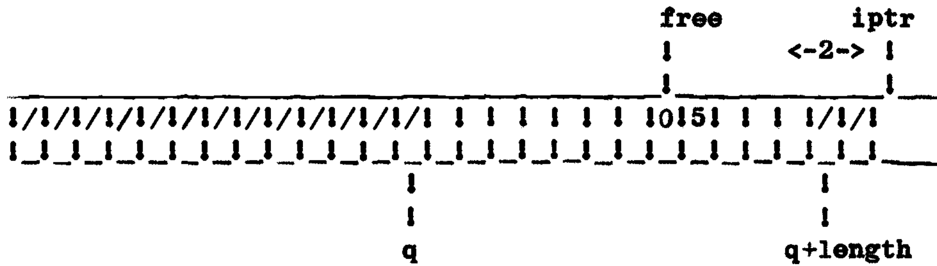
6) The last block (5) must be moved up. Set POSN(p)=iptr and ADD(p) to length. Then copy the values between q+length and iptr-1 into the locations beginning at q.

7) Reduce iptr by the value of length.





length=7



length=12

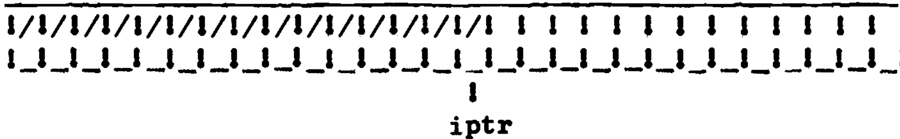


Fig. 7

There remains the second part of the garbage collection, i.e. the resetting of the pointers. Pointers to INFO are held in four arrays: VAL, CAR, CDR, and PARAM. (PARAM is used in setting up parameters. It is necessary solely for the purpose of altering pointers in the event of a garbage collection during the processing of an actual parameter.)

The routine Empty, local to garbage collect, is called for each of these arrays. Here is the algorithm for ´empty´.

1) Examine the first element, a, of the array.

2) Set m=1

3) If a<POSN(m), reduce the pointer by the value ADD(m) and go to (5).

4) Go to (3) if there is another entry in POSN.

5) Go to (2) if there is another element, a, in the array.

9.3 The Organisation of ALG

The garbage collection for INFO relies on the fact that only four known arrays, all quite small, can contain pointers to INFO. Therefore the adjustment of these pointers at the end of the garbage collection is not too lengthy. The situation with ALG is different. So far ALG has been used in two different places: for storing labelled statements, and for storing the value of a variable if it is an algebraic expression. This latter use is sufficient to make the method of searching for all pointers to ALG impractical, since searching INFO would involve looking at 5000 entries.

Another use of ALG makes it impossible. The routines used for obeying algebraic commands require a large amount of local space. In order to optimise the use of this space, the technique of claiming space from ALG is used, rather than declaring local arrays. This ensures that the space can be claimed at run time, and then only if necessary. Since all the algebraic routines are mutually recursive, this can save a good deal of space. Because of this use of ALG, garbage collection for ALG is required quite frequently, whereas the garbage collection routine for INFO is not used very often.

To access ALG, two pointer cells are used, ARNI and ARNJ. Together they are said to form a set of ARN cells.

ARNI points to the beginning of the expression (or statement) stored in ALG, and ARNJ points to the end. Once an ARN cell has been claimed for a particular use, it cannot be freed until the garbage collection for ALG (a routine called Alg Collect) is called.

If space is needed on ALG, the function Get(lth) must be called, where lth is the amount of space needed. The algorithm for Get is as follows.

1) Look at ARNI(arnp) (where arnp is the pointer to the ARN cells)
2) If this is zero, this cell is free. Aptr points to the next free position in ALG. Call Alg collect if lth+aptr>5000. Then set ARNI(arnp) to aptr, ARNJ(arnp) to aptr+lth-1. Set aptr = aptr + lth, and arnp = arnp + 1. The result is the number of the ARN cell just filled. (Alg Collect faults the program if ALG is full.)
3) If ARNI(arnp)#0, increase arnp by 1. If this is greater than 500 (the size of the ARN list), set arnp to 1. Go to (1) unless all the ARN cells have been examined this time. Fault the program

Thus a cyclic search is made for a free ARN cell. Once this has been found, its position, k, can be stored if necessary, and the required statement or expression is copied into ALG, beginning at ARNI(k).

The three processes that use ALG store k in the following ways.

1) For a labelled statement, another array, STATE is used to store k. Thus if the position of the statement in the Storage Tree is j, the STATE(j) has the value k.

2) For an algebraic expression stored as the value of a variable, INFO(j+1) = k, where j points to the beginning of the description of the variable.

3) For space claimed to be used locally by an algebraic routine, the IMP statement I==ARNI(K) is used. I is declared to be a %SHORTINTEGERNAME variable. Then setting I==ARNI(K) has the same effect as using a name type parameter; whatever alteration is made to I in the routine is in fact done to ARNI(K). This means that if ARNI(K) is altered by the garbage collection, since I accessed the position of ARNI(K), rather than merely having its value, I will still point to the correct position in ALG.

If some expression in ALG can now be destroyed, the first entry in it (i.e. that which ARNI points to) is set to 255. No other alteration is made until Alg collect is called, and in particular, the ARN cell is not freed. Fig 8 shows a typical situation before a garbage collection takes place for ALG.

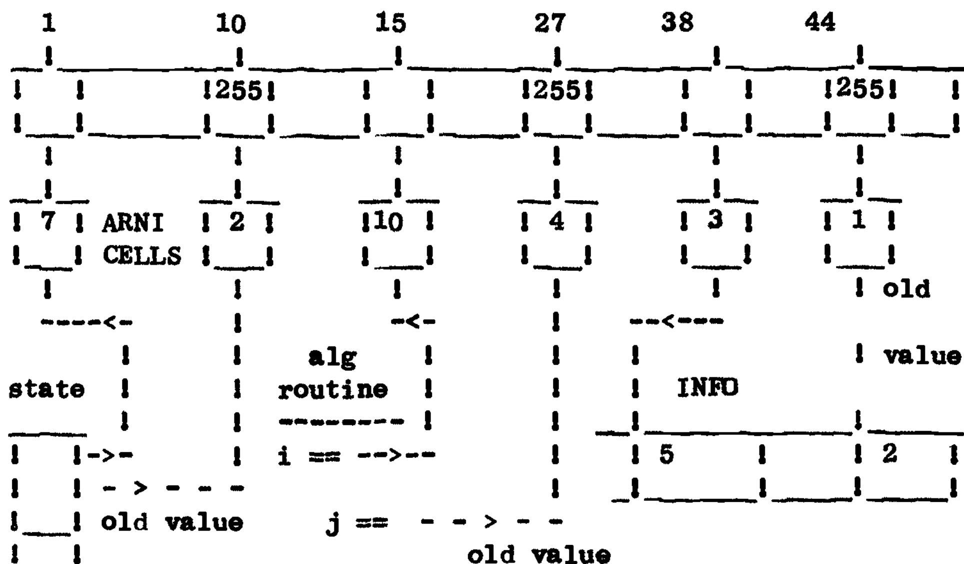


Fig. 8.

The situation after garbage collection will be

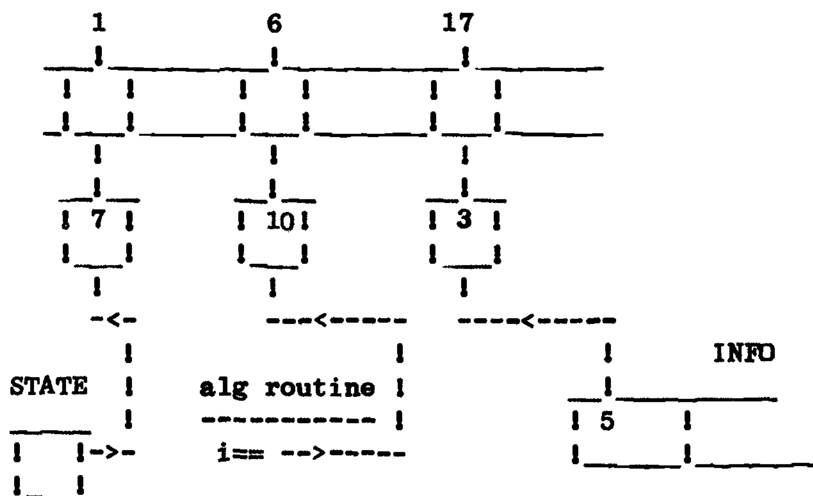


Fig. 9

ARNI(1) = 0

ARNI(2) = 0

ARNI(4) = 0

It will be noticed that in the diagram of the state before garbage collection, the ARN cells are not in order. Initially, of course, the first ARN cell will be claimed,

and will point to the beginning of ALG; the next will be the second which will point to the next block of elements, as in Fig 10.

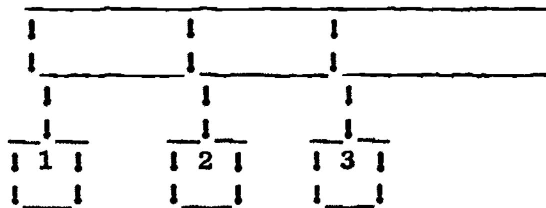


Fig 10

However, once a garbage collection has been made, this order will not necessarily be found the next time Alg Collect is called. Therefore the order cannot be assumed.

The examination of ALG must begin at its lowest end. Therefore, returning to Fig 8, the position in ARNI(7) must be examined first, then ARNI(2), then ARNI(6), etc..

Hence Alg collect first finds the order in which the ARN cells are to be examined. A local array, PNT, is used. Cycling through the ARN cells, a new cell of PNT is set pointing to the ARN cell, provided ARNI is not zero, thus giving the situation shown in Fig. 11.

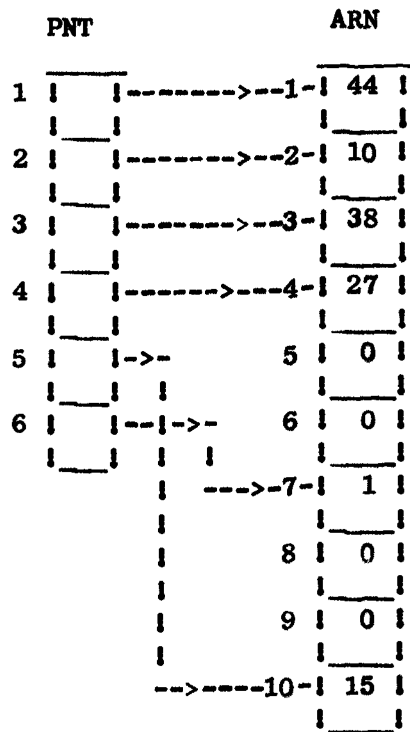


Fig. 11

A sort routine is used, to let the pointers of PNT give the ordering of the contents of ARNI, as shown in Fig. 12.

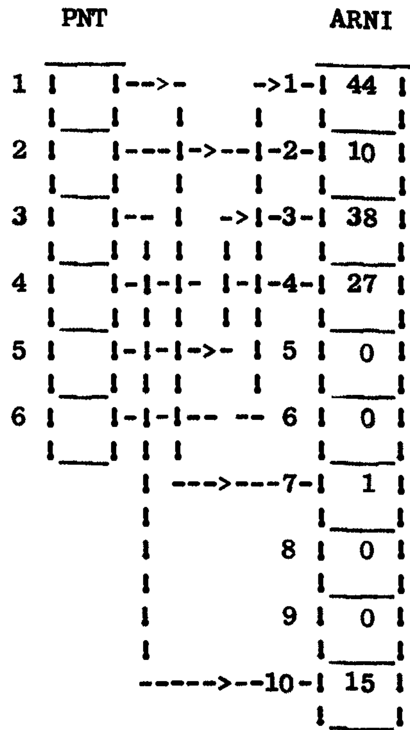


Fig. 12

Now, cycling for i=1,1,p, where p is the number of entries of PNT, the following algorithm is obeyed. (Lth is initially zero.)

1) Set J=PNT(I), K=ARNI(J), L=ARNJ(J).

2) Go to (4) unless ALG(K)=255.

3) Add (L-K+1) to Lth. Set ARNI(J) to zero. Finished.

4) Set ARNI(J) to K-Lth, ARNJ(J) to L-Lth. Copy ALG(K) to ALG(L) into ALG(K-Lth) to ALG(L-Lth). Finished.

Once this has been done p times, aptr is reduced by the value Lth. This completes the garbage collection for ALG.

X Implementing AML - Polish Notation and Expressions.

The analysis record of an expression, although an improvement over the recursive definition that is used in IMP, is not suitable for processing algebraic expressions. The applications of an algebraic command to an expression requires that the expression should be broken up into different parts, and these parts are then reassembled in a different order. The precedence of operators in these operations is of course, very important. After some attempt at processing expressions in an infix form, it was found that a variety of prefix Polish notation was most useful. (A discussion of the reasons for choosing this notation is given in Chapter 12.)

The five arithmetic operators are

**, /, *, - and +.

Of these, ** and / are definitely binary. However, * and +, being commutative and associative, can be regarded as n-ary operators. Thus

x**y

is written in Polish notation as

Exy   (E = exponentiate)

and

x*y*z

is written as

*(x,y,z)

There must be a way of delimiting the operands of an n-ary operator. Brackets are a useful notation for describing Polish expressions. However, in the internal representation the sign ´*´ is followed by two bytes which together give the total length of the list of operands. This enables the interpreter to find the end of an expression easily.

10.1 Polish Notation

Three types of elementary operands may occur in the Polish representation of an expression. They are a constant, a name, and a name followed by a list. The first is represented by either five or nine elements of a byte array. Integers and real numbers require four bytes for their value, and one for the code. Rationals and long reals require eight bytes for their value, and one for the code. The first element will hold the code, which is a number between 1 and 4. A name is represented by two byte array elements, the first containing the number 5, and the second the hash code of the name required.

The third type of operand is represented by the number 6 followed by the hash code of the name. The next two bytes contain the number of bytes needed for the description of the list. Two bytes are needed as this list may be greater than 255 in length. The following element gives the number of primes that follow the name, and the rest of the description consists of the subscripts given as expressions in Polish notation. For example, if the hash code of ´a´ is

20, and that of ´b´ is 21, the Polish representation of ´a(b,3)´ is

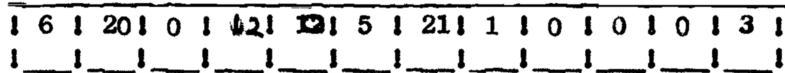


Fig. 1

Expressions are built up by prefixing a number of elementary operands by an operator. If we define an expression to be an elementary operand, we can build up expressions of any complexity. There are two binary operators ´/´ (divide) and ´E´ (exponentiate). ´E´ was chosen to represent exponentiation in the internal representation since the two more usual forms are not suitable. Up-arrow is not available in the character set of some equipment, and ´**´ takes up two locations. Thus the expressions a/b and ´a**b´ are represented in Polish notation as

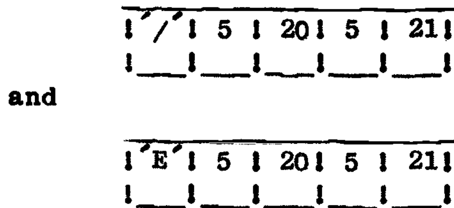


Fig. 2

The operators ´*´ and ´+´ may have any number of operands, and two bytes following the operator contain the length of the operands pertaining to this operator. Thus ´a+b+c´ and ´a*b*3´ are represented as

´+´	0	6	5	20	5	21	5	22

where 22 is the dictionary number of ´c´.

and

´*´	0	13	5	20	5	21	1	0	0	0	3

Fig. 3

The minus operator ´-´ is used as a unary operator only. Negative numbers are stored as themselves, and not as ´-´ followed by a positive number. For example, the binary form of -3 is

1111 1111 1111 1111 1111 1111 1111 1101

and stored in a byte array, it becomes

255 255 255 253

Hence the representations of -3, -a, a-3 are

1	255	255	255	253

´-´	5	20

and

´+´	0	11	5	20	1	255	255	255	253

Fig. 4

Now we shall give some more examples of general expressions.

1) ´a+b*c´

´+´	0	9	5	20	´*´	0	4	5	21	5	22

2) ´a*(b+c)´

´*´	0	9	5	20	´+´	0	4	5	21	5	22

3) ´a-b´

´+´	0	5	5	20	´-´	5	21

4) ´a+b/c´

´+´	0	7	5	20	´/´	5	21	5	22

5) ´a**b/c´

´/´	´E´	5	20	5	21	5	22

Fig. 5

10.2 The Routine Evalexpr

This routine takes the analysis record of an expression and transforms it into Polish notation. Its function divides into two distinct steps.

1) Find the current value of each operand.

2) Form the expression in Polish notation, performing numerical calculation wherever possible.

As mentioned earlier, this routine works on all expressions, numerical and algebraic. It is the need to check for possible numerical calculations, and also to order algebraic expressions that makes it fairly complicated. The first part of the routine, i.e. finding the current value of

an operand, is the simpler of the two, and so will be discussed first.

The operands that may occur in an expression are

i) (LABEL EXPR)

ii) (CONST)

iii) (NAME) (LIST)?

iv) '('(SELECTOR)? (EXPR)')'

The algorithm for this part of the program is

1) Find out which alternative of (OPD) is being processed.
2) If it is a (LABEL EXPR) (i), evaluate the label, get the statement at that label, analyse it, and call Evalexpr to put this into Polish notation. Put the result into the array Z.
3) If it is a (CONST) (ii), copy it into the array Y.
4) If it is alternative (iii), check to see if (NAME) is an algebraic constant. If so, go to (5). If it is the name of an array, go to (6). If it is a function, go to (7). Copy the value of (NAME) into Y if it is numeric, and into Z if it is algebraic.
5) If there is a list, call Evalexpr for each of its elements. Copy (NAME) and the processed elements into Z.
6) Calculate the position in the array, and find the value stored there. If it numeric, copy it into Y; otherwise copy it into Z.

7) Evaluate the function. Put the result into Y if numeric, and Z otherwise.

8) If the alternative is (iv), evaluate the expression using Selectq. This is a routine which performs the selection (if any) on the expression, and will be discussed in chapter 13. Put the result in Y or Z as necessary.

Hence at the end of this process, the array Z will hold the value of the operand, if it algebraic, and the array Y will have the value if it is numeric.

An expression that has n operands will cause a cycle to be executed n times. Inside the cycle, the value of the operand is obtained by the method described above. Let this value be A. Then the operator op1 preceding A, and the operator, op, following it are used to determine what action is taken for A. Three arrays are used to store the operands. They are:

1) R, which holds terms that are added together, and therefore ultimately has the final result.

2) S which holds terms that are multiplied together.

3) T which holds terms connected by ´E´ or ´/´.

Initially R, S and T are all empty.

For the first operand, op1 is taken to be ´+´, and for the last operand, op=´+´. Each operand may be dealt with in one of several ways, depending on the circumstances that govern its use. These are mainly

i) The values of op and op1

ii) Whether A is numeric, and whether there is another operand that is also numeric and which can be combined with A.

The various possibilities are listed here, taking the simplest cases first.

1) op1=+ and op=+

Example (i) a+b

The final result for this will be +(a,b).

Example (ii) 3+4

The final result for this will be 7.

The term A that is being considered must be put in R. If A is numeric, and if the last term, B, in R is numeric, the total B+A replaces B in R. Otherwise A is copied to the end of R. In pseudo-IMP this can be written

```
A=B+A %if R->R.C %and num(B) %and num(A).
R=R.A
```

(where R->C.B means B is the last term of R, C is the rest, and C may be empty. Num(B) is a boolean test giving the result TRUE if B is numeric.)

2) op1=* and op=*

Example (i) a*b*c

which gives the result *(a,b,c).

Example (ii) 3*4*c

which gives the result *(12,c)

A is put in S, in the same way that it was put in R in (1), i.e.

A=B*A %if S->S.B %and num(B) %and num(A)

S=S.A

3) op1=+ and op=*

Example a+b*c.

which gives a result +(a,*(b,c)).

S must be empty at this stage, so S=A.

4) op1=* and op=+

Example b*c+d

which gives a result +(*(b,c),d)

A is put in S, as in (2). Then S is treated as an operand to be put in R.

A=B*A %if S->S.B %and num(B) and num(A)

S=S.B ; *(S)

S=S+B %if R->R.B %and num(B) %and num(S)

R=R.S ; S=´´

(where *(S) means insert a ´*´ in front of S if S has more than one term.

i.e. S=a,b gives S=*(a,b)

but S=a is unaltered.)

5) op = -

If op=-, A is treated as if op=+. Then before the next operand is processed, op is set to + (so that op1 will be + for the next operand) and the variable ´minus´ is set to 1. Whenever minus=1, (1) is modified to

%if R->R.B %and num(B) %and num(A) %then A=B-A %else A=*(-1,A)

R=R.A

The cases listed above cover all the cases of expressions not involving / or E. When these operators are used, a number of different cases arise. These will be listed below in increasing complexity. For any particular case, all cases above it are assumed false.

The first case is simple:

6) op1 = + or * and op = E or /

In this case, T is empty, so

T=op.A ; op2=op1

Cases (7) to (16) assume that both op and op1 are E or /.

7) op1=/ and op=E

Since op has a higher precedence than op1, A must be copied into T.

T=T.´E´.A

8) op=/ and ¬num(A)

Example b*a/3

which gives the result /Eba3

Considering the operand ´a´, T is changed from

Eb to /Eba.

or, in general, T=´/´.T.A

9) op=´E´, ¬num(A) and T#´/´.B (i.e. the first entry in T is not ´/´.)

Since case (7) is not true, this also means that op1=E.

Example b**a**c

which gives the result EEbac

Considering the operand ´a´, T is changed from

Eb to EEba

or, in general, T=´E´.T.A

10) num(A) and T=opp.B and num(B). (T consists of an operator followed by a numerical value.)

The possibility that opp=´/´ and op=´E´ has been elimated by (7). Therefore the calculation ´B.opp.A´ can be performed. This is put back in T, preceded by op.

i.e. A=B**A if opp=´E´

A=B/A if opp=´/´

T=op.A

11) op1=´/´ or T#´/´.B

The tests listed above have eliminated quite a few cases.

a) If op1=/, we can deduce

i) op=/ (from 7)

ii) num(A) (from 8)

iii) ~(T=/.B and num(B)) (from 10)

An example that satisfies these conditions is

c**d/3/b

which gives the result //Ecdb3

Considering ´3´, T is changed from

/Ecd to //Ecd3.

b) If T#/.B, we deduce

i) num(A) (from 8 and 9)

so ii) T='E'.b and ¬num(b) (from 10)

An example satisfying this is

c**3**b

which gives the result EEc3b.

Considering '3', T is changed from

Ec to EEc3. Taking the general case for both (a)

and (b),

T=op.T.A.

The cases left after test (11) are

i) op1=E, op=E

ii) op1=E, op=/, A numeric

iii) T=/.B

These can be deduced in the following way.

a) op1 = E (from 11)

b) T = /.B (from 11)

c) If op=/, then num(A) (from 8)

The situation where op1 =E, but T=/.B arises from a case like the following.

T = a/c**d**e

Processing each element in turn,

a gives /a

b gives /aEc

d gives /aEEcd

It is the processing of d that is to be considered just now. The position in which the new 'E' is to be inserted is determined by the extraneous E already there. There may in

fact be other exponential signs there (e.g. T=/EabEc), but only one will not be linked to two operands. Let us split T to be

T = /X.E.Y

12) ~num(A)

The example given above satisfies this. From (i), op must be E.

Therefore T = /.X.EE.Y.A

13) num(Y) and op=E

Example a/3**2**e

which gives /aE9e

Considering '2'. T changes from

/aE3 to /aE9.

In general, A=Y**A; T=/.X.E.A

14) num(Y) and num(X) and op=/

Example 7/3**2/e

which reduces to /3e

Considering '2', T changes from

/27E3 to /3.

In general, A = Y**A

A = X/A

T = '/'.A

15) num(Y) and op=/

Example a/3**2/e

As (14), but ~num(X)

A=Y**A

T = //X.A

16) op1=/ and ~num(Y)

No reduction possible. Do (12) if op='E', and (13) if op=/.

The remaining examples deal with the case where op = + or *, and op1 = / or E.

17) ~num(A)

A = T.A

18) T = opp.B and num(B)

A = B**A if opp=E

A = B/A if opp=/

19) op=/ or T#/.B (See 11)

A = T.A

In the remaining cases, let T = /.X.E.Y

20) num(Y) and ~num(X)

A = Y**A

A = T.A

21) num(Y) and num(X)

A = Y**A

A = X/A

22) ~num(Y)

A = T.A

After T has been examined and emptied, the result is put in A, to be treated as an operand. Op1 is set to op2 (the operator occurring before T), and tests (1) to (5) are applied. Finally, when all the operands have been examined, the required expression is given by +(R). (Where +(A) is defined in the same way as *(A).)

The cases listed above exclude the logical operators. However, these fit into the description very easily, as follows.

1) >> and << - treated as **

2) & - treated as *

3) ! and !! - treated as +

It was felt that including them in the description would serve no useful purpose; the rules would be less easy to follow, and no significant changes are made for them. The program is faulted if logical operators are not applied to integers. Hence they will be eliminated from expressions by Evalexpr. The algebraic routines do not recognise logical operators.

10.3 Evaluating Label Expressions.

An integer function called Getlabel processes the label expressions, returning the position found in the Storage Tree as its result. Getlabel also processes the label of a labelled statement. A parameter ins is used as a flag, and may take three values:

0 Fault the program if the label required is not in the Storage Tree.

1 Insert a cell for the label if it is not already there.

2 The label of a stored statement (rather than a label expression) is being processed.

If there are n components of the label, the following is done n times.

1) Evaluate the component. If ins = 2, this is a matter of copying the value of the next entry in PL into a variable nm. Otherwise the type of the component must be considered.

   a) A constant - Check that it is an integer, and put it in nm.

   b) An expression - Call Eval-to-int, which applies Evalexpr and checks that the result is an integer.

   c) A name - This may be a variable which holds an integer, or, inside a routine, the name of a label parameter. In the latter case there may only be one component. The value of the label parameter is returned as the result. Otherwise the name is checked for an integer, and its value is put in nm.

2) Find the position of the component.

A variable, ptr, is passed as a name type parameter to the routine. If the label is a full label, initially ptr is 1. Otherwise ptr names the cell below which the required position will be found. Fig. 6 illustrates this, for the second label expression of the statement

```
%do 3:1, :3
```

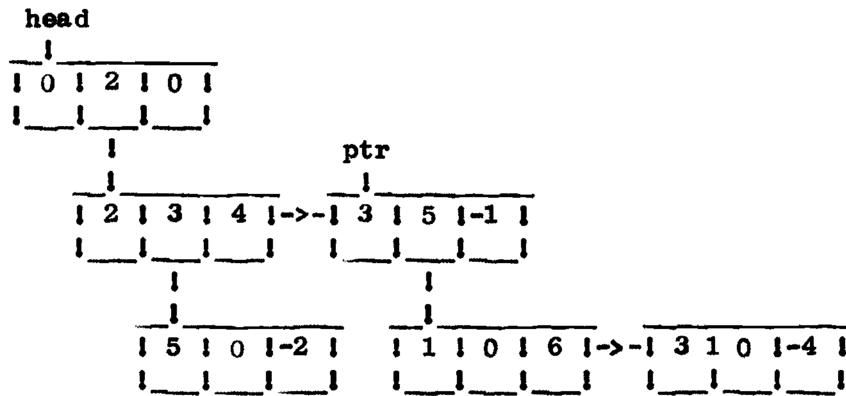


Fig. 6

Since the abbreviated label ´:3´ means 3:3, the pointer points to cell representing the label 3. After the first component has been processed, ptr is set to point to this position for subsequent components. For example, in the processing of 3:1, ptr = 1 initially, but for the second component, ptr will have the position shown in the diagram.

The steps taken to find the position of nm are as follows.

i) Set j=BELOW(ptr).

ii) If j=0, there are no cells below ptr. Therefore fault the program if ins = 0. Otherwise insert a cell, k, below ptr. If this is the last component, the result is k. Otherwise set ptr = k and proceed to the next component.

iii) Set k = 0, and do (iv) while NO(j)<nm and j>0.

iv) K = j; j=AFTER(j). This tracks along the

cells until it finds one whose entry in NO is not less than nm, or until it reaches the end of the list (when j is negative).

v) If (iv) stopped because NO(j)>=nm, check for equality. If NO(j) = nm, the required cell has been found. Hence this is the result if the last component is being processed. Otherwise set ptr = j, and proceed to the next component.

vi) If NO(j)>nm, the cell is not there. Fault the program if ins = 0. Otherwise insert a cell between k and j (hence the reason for preserving k). This new cell is the required cell. Either give it as the result, or set ptr to it, as described in (ii) and (v).

vii) If (iv) stopped because j<0, no cell matching nm has been found. Hence proceed as in (vi).

This completes the description of the routine Getlabel.

10.4 Conditions

Three kinds of conditions are available in AML:

1) Simple conditions (which include bracketed conditions).

2) %and-conditions

3) %or-conditions.

Here is the algorithm for dealing with a general condition.

1) Consider the first simple condition.

2) Evaluate it, setting j to 1 if it is TRUE, and to 0 if it is FALSE.

3) Return if there is no other simple condition following.

4) Return if j=0 and the condition being processed is an %and-condition.

5) Return if j=1 and the condition is an %or-condition.

6) Get the next simple condition and go to (2).

The evaluation of the simple conditions not involving Patterns is straight forward. The two conditions

(EXPR) ´%matches (PATTERN)

(EXPR) ´%contains´ (PATTERN)

will be discussed in the Chapter on patterns. (Chapter 13). The remaining alternatives are

1) ´(´(CONDITION)´)´

2) (EXPR)´//´(EXPR)

3) (EXPR) (COMP) (EXPR) (RESTCOMP) ?

where (RESTCOMP) = (COMP) (EXPR)

6) (LB).

Here is the algorithm for the routine processing these alternatives.

1) Get the alternative.

2) If it is the 1st, call the routine Condition recursively.

3) If it is the 2nd, evaluate both the expressions. Fault the program if they are not both integers, m

and n. Set j=1 if fracpt(m/n)=0, and j=0 otherwise.

4) If it is the 3rd, evaluate the first two expressions into P and Q. If (COMP) is '=' or '#', P and Q may be algebraic. Compare them element by element, setting j to the correct value.

5) For (COMP) not '=' or '#', P and Q must be numeric. Fault the program if they are not. If (COMP) is '<' or '<=', test P<Q, otherwise test Q<P. This is done by the integerfn Lessq(a,b) which finds the type of a and b, converts them both to reals, and returns the result 1 if a<b, and 0 otherwise. Set j to Lessq if there is no (RESTCOMP).

6) Set the (RESTCOMP) pointer to zero. Copy Q into P, and evaluate the last (EXPR) into Q. Fault the program if the comparators are incompatible. Go to (5).

7) If the alternative is the 6th, evaluate the label expression. Analyse the statement there as a condition. Then call Condition for the result.

XI Implementing AML - the Commands.

The commands of AML are also processed by Exec. The syntax of a command statement, which is an (UNCONDST) is

(COMMAND)*(UNTILCL)?.

Hence a number of commands may be processed in one statement, and may be governed by an (UNTILCL). The syntax of the latter is

```
%until (CONDITION) !
%while (CONDITION) !
%for (NAME) = (EXPR) , (EXPR) , (EXPR).
```

Examples of command statements are

```
%do 1
%do 1:1 %while i<0
%simplify A(i) %for i=1,1,10
%distrib A(i),%simplify B(i) %until i>5
```

The process for obeying a command statement is as follows

1) See if there is an (UNTILCL). Skip to (6) if there is none.
2) Let a be the alternative of the (UNTILCL); a=1 for %until, a=2 for %while, and a=3 for %for.
3) If a=1, no test should be made until after the commands have been obeyed.
4) If a=2, the commands are only obeyed if the (CONDITION) is true. Therefore call Condition to test it. If it is false, the process is finished.

5) If a=3, the %for loop conditions must be set up. The three expressions are evaluated into p, q, and r. They must all be integer valued. Then NAME is given the value p.

6) The list of commands is examined and processed. This is described in detail below.

7) The processing of the statement is finished if there is no (UNTILCL).

8) If a=1, the (CONDITION) is tested. Go to (6) if it is FALSE. Otherwise the process is finished.

9) If a=2, the (CONDITION) is tested. Go to (6) if it is TRUE. The process is finished otherwise.

10) For a=3, the process is finished if NAME>r and q>0, or NAME<r and q<0. (A more general condition than NAME=r.) Otherwise add q to NAME and go to (6).

Commands can be divided into two kinds: general and algebraic. The general commands will be discussed first. They are

1) %do (LB)*

2) %write (LB)*

3) %erase LB)*

4) %label (LB)* %as (LB)*

5) %print %results (LB)*

6) %print %no? %results

7) %read (NAMELIST)* %from %file (EXPR)

8) %eval (NAMELIST)*

11.1 %do, %write, %erase, %label

The first four are processed together; the last four are examined individually. This arrangement is made because the (LB)*s of the first four commands can be evaluated before the commands are obeyed, and stored in an array local to Exec. Obviously it is advantageous to have one piece of program to do this. For the %do command it is imperative that the evaluation of the label expressions is done first, since the statements that %do activates may change the variables used in (LB)*. For the other three commands, it is immaterial whether the label expressions are evaluated first, or as required. Here is the algorithm for obeying the first four commands.

1) Get the first label expression.
2) Evaluate it, storing the position obtained in the array PREV.
3) Go to (2) if there is another label expression.
4) Jump to the switch label appropriate to the current command.
5) %do: Call the routine Execute for each entry in PREV. Execute calls Exec for each statement stored in the sub tree given by its parameter. If '%exit' or '%return' is executed (legitimately) before all entries in PREV have been used, do not execute the remainder.
6) %write: Call the routine Printtxt for each member of PREV. Print txt prints all statements occuring

in the sub tree given by its parameter.

7) %erase: For each entry in PREV, remove all statements stored in the sub tree it defines. This is done by storing an empty statement at the relevant label.

8) %label: do (9) to (18) for each member of PREV.

9) Let the member of PREV be I.

10) Evaluate the next label expression of the second (LB)* into J. Fault the program if there is none. Set K=I.

11) Copy the statement at I into position J of the Storage Tree. Go to (14) if I has no statement below it.

12) Set I=BELOW(I).

13) If there is not a cell below J whose NO is the same as NO(I), insert one, call this cell J and go to (11).

14) If I=K, this is the original label, so the copying is complete. Go to (17).

15) Set I=AFTER(I). Go to (16) if I<0. If there is not a cell after J whose NO is equal to NO(I), insert one, calling it J. Go to (11).

16) Set I=|I|, thus tracking back up the tree. Go to (14).

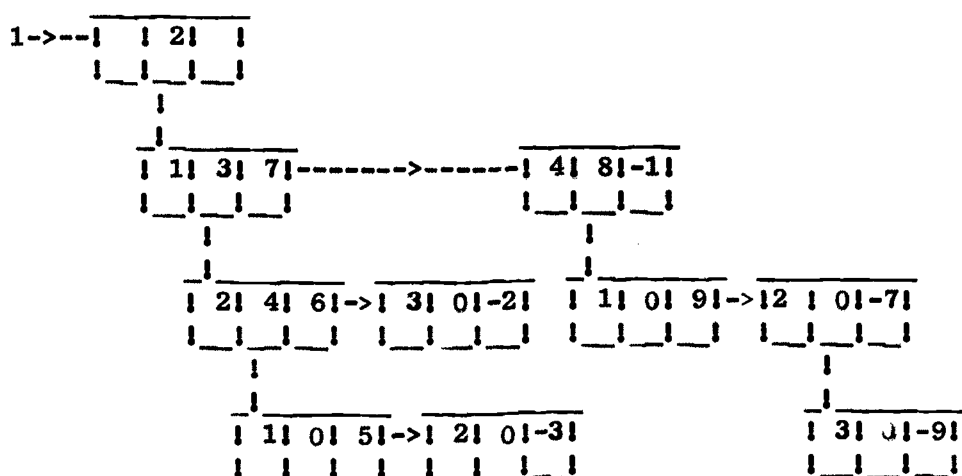
17) Go to (9) if there is another entry in PREV.

18) Fault the program if there are any label expressions left in the second (LB)*.

Steps (9)-(18) of this algorithm give a variation of the technique that is used for all processes involving blocks of statements. %Do, %write and %erase use similar algorithms. Fig. 2 describes the situation when the statement

%label 1:2,4 %as 2:3,3:1

has been executed, with the Storage tree initially as in Fig. 1.



1:2 a=1

1:2:1 b=3

1:2:2 c=5

4:1 d=2

4:2:3 e=7

Fig. 1

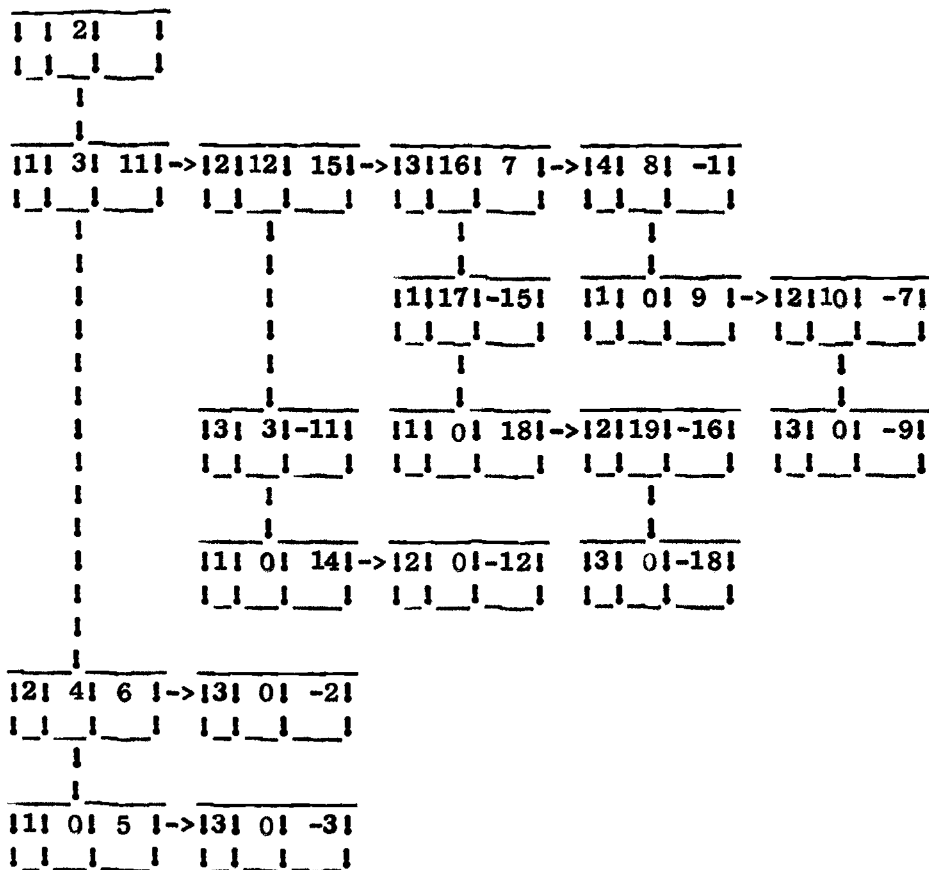


Fig. 2

1:2 a=1 2:3 a=1

1:2:1 b=3 2:3:1 b=3

1:2:1 c=5 2:3:2 c=5

4:1 d=2 3:1:1 d=2

4:2:3 e=7 3:1:2:3 e=7

11.2 %print %results (LB)*

%print %no? %results

The purpose of these commands is to supress unwanted printing of the results of algebraic commands. Hence the information must be stored in a global array called PR. It and a variable ppr are used to hold the information, which is organised as follows.

The default condition is that all results are printed out. This is indicated by setting ppr to zero. %Print %no %results suppresses all printing, and this is indicated by ppr having the value -1.

%Print %results (LB)* requires that results are only printed for the arguments that are given in (LB)*. This statement is processed by evaluating each member of the list and storing the position in the Storage Tree that is obtained in PR. Ppr is set pointing to the last entry in PR. %Print %results sets ppr back to zero, thus enabling all printing.

11.3 %read (NAMELIST)* %from %file (EXPR).

This command is used to read data from a file. When AML is called certain files may have been given stream numbers under the EMAS system. Hence (EXPR) is evaluated, (it must give an integer), and this number specifies which file the data is to be taken from. The IMP routine Selectinput is called, with this number as its parameter, thus causing the input to be taken from the correct file.

In order to discuss the process from here, it is necessary to describe briefly the organisation of stream files in IMP. Once Selectinput has been called, any read instruction will take its data from the file selected. The input in the file is regarded as being in card images; a newline ending the card. Data can be read in single numbers, but when the file is closed (by selecting another file, for example), and then re-opened, the input will be read from

the next card image. So it is possible that some information may be lost. For example, if the data in file 1 is

```
2 3 4 5 6 7
8 9 10
```

Then the IMP program

```
SELECT INPUT(1)
READ(I);READ(J)
SELECT INPUT(0)
..........
SELECTINPUT(1)
READ(K)
```

will result in I having the value 2, J having the value 3, and K the value 8. The numbers 4, 5, 6, and 7 will be lost.

It was felt that this restriction should not be passed on to AML. Hence any numbers at the end of a card image that are not used by a %read command should be saved, and held ready for use if another %read command should be executed for that file. Only the first 72 characters on a card are read by an IMP program. Since the contents of a file are treated as card images, any character after the 72nd in a line will be ignored. This restriction is carried through to AML. Hence, for each file there may be up to 72 characters outstanding when a %read command is finished. It was decided to allow up to six input files to an AML program, apart from the console, which is always regarded as file 0. Thus a maximum of 432 characters must be stored. They are put in a byte array called FILE. Whenever the read command is

executed, a check must be made to see if any characters have been stored in FILE for the particular file selected. If so, the value is read from there; otherwise it is read directly from the file.

Data to the %read command may be numbers or algebraic expressions. They are separated from each other by a space. The actual reading is done by the routine Readsym. This has an %own byte array L which holds the position in FILE from which the next character is to be taken. This is described in Fig. 3.

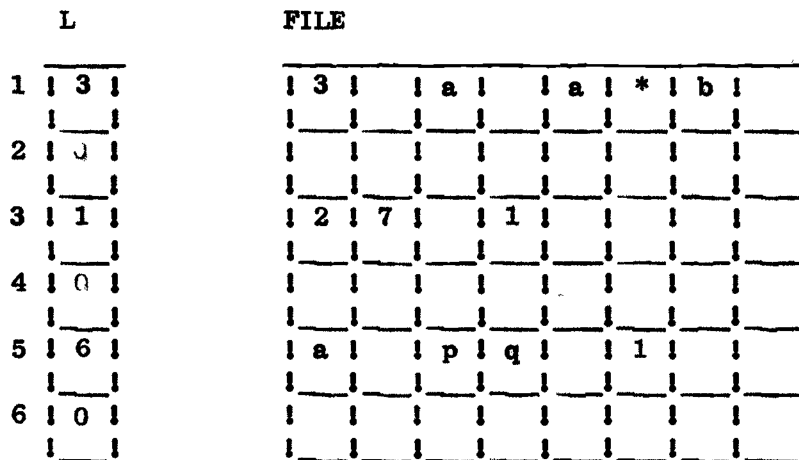


Fig. 3

In this diagram, files 2, 4, and 6 have no characters stored for them. The next character to be read from file 1 is in FILE(1,3); the next character to be read from file 3 is in FILE(3,1) and the next for file 5 is in FILE(5,6).

Here is the algorithm for Readsym, which has a name type parameter t.

1) Do (3) if FILE is non-empty for this input stream.

2) Readsymbol(t) and return.

3) Set t=FILE(j,L(j)), where j is the number of the input stream.

4) If t=newline, set L(j) to 1; otherwise increment L(j).

5) Print t. (To let the user see what has been read in). Return.

The same routine is used for reading data as for reading statements. However a value type parameter, r, is set to 0 for reading statements, and to 1 for reading data. The differences these values of r cause are:

R=0

a) Suppress reading from FILE in readsym.

b) Exit when newline is encountered.

R=1

a) Supress search for label.

b) Exit when a space or newline is encountered.

In both cases, the input is put into the array AR.

There follows the algorithm for the command %read.

1) Evaluate (EXPR) into I.

2) Select input (I).

3) Get the first name, N, which is the name of a scalar or of an array element.

4) Call the read routine with r=1, putting the result in the array Z.

5) Analyse Z as an expression, applying Evalexpr to the result.

6) Copy the expression into N.

7) Go to (4) if there is another member of (NAMELIST)*.

8) Go to (10) if the last symbol was a newline. Set L=1.

9) Read symbol(t), and put t in FILE(I,L). Increment L. Go to (9) unless t was a newline.

10) Call Select-input for the file that was being used before.

The last step requires some explanation. The routine Getfile enables users to read statements from a file. It may be that one of these statements is a %read command, and therefore when it is finished, statements should still be read in from this file, and not from the teletype. The integer slct is used to list these file numbers. Since files are numbered from 0 to 6, three bits are used for each number. Therefore ten file numbers can be stored in the integer slct. When a new file is used, its number i is calculated and stored in slct by the IMP statement

```
SLCT = SLCT<<3!I.
```

Then when the old file number is to be restored, this value is removed. The statements to do this are

```
SLCT = SLCT>>3 ; ! REMOVES LATEST VALUE
I=SLCT&7 ; ! LAST VALUE
SELECT INPUT(I)
```

Initially, of course, slct is zero.

11.4 %eval (NAMELIST)

Each argument of %eval is the name of either a variable or an element of an array, which must have an algebraic expression as its value. The routine Eval is called, and this examines each operand of the expression in turn. The algorithm for testing each operand is as follows.

1) If the operand is numeric, copy it into the array R which holds the result.
2) If the operand has the same name as the argument (e.g. as A = A + B), copy the name into R.
3) If the operand is a scalar (i.e. if there is no (LIST)), set k = VAL(h), where h is the hash code of the operand.
4) If k = 0, the name is an algebraic constant. Therefore copy it into R. Otherwise copy the value of NAME into R.
5) If the operand is of the form NAME (LIST), call Eval for each element of the list. If NAME is an array, get the value (if any) of the element that is referenced.
6) If the name is a function, call it, using the evaluated list as parameters, and put the result in R.
7) Otherwise copy NAME into R, followed by the evaluated list.

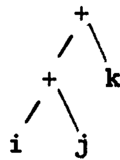
Hence R, which is constructed with the same operators as the original expresssion, will hold the result of

evaluating the argument A of %eval. This expression is stored as the new value of A.

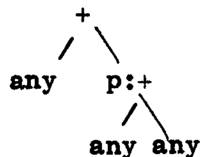
XII Implementing AML -The Algebraic Commands

The facilities whose implementation has been described so far are those which have a parallel in IMP. Differences occur because of the interactive nature of AML, and the reasons for these differences have been given. The rest of this description will be concerned with the Algebraic manipulation part of the language, and here there is no precedent that can be closely followed. In Chapter 10 a description of Polish expressions was given. Before proceeding further it is necessary to justify the choice of representation and to discuss alternative ways of representing algebraic expressions.

Experience with Formula Algol led to the belief that formulae should not be stored as binary trees. The expression ´i + j + k´ is stored in Formula Algol as



the pattern ANY + p:(ANY + ANY) is stored as



This means that the expression

i + j + k == ANY +p:(ANY + ANY)

is said to be FALSE, in contradiction to what one would expect.

Further consideration of the subject led to the belief that in general the operators + and *, being associative and commutative, are regarded as applying to each of their operands in equal strength. One does not think of (i+j)+k, or yet i+(j+k), but knowing that these two expressions are equivalent, we regard all three operands as having equal status.

Again, a binary representation of expressions makes recognising equality more complicated. (a+a)+b would easily collapse to 2*a+b, but (a+b)+b requires a different approach. All manipulation of algebraic expressions depends on the ability to recognise the equality of two expressions. With the operators + and * this is a considerable problem. The two expressions

a*b*c + p*q*r + x*y*z

q*r*p + x*z*y + c*a*b

are equal; but a fair amount of testing of various combinations is necessary before this can be discovered.

Algebraic manipulation by its nature is highly recursive. It seems that a method which reduces this recursion to a minimum is necessary. Binary trees do not do this.

Having decided against a binary representation, what remains? An attempt was made to process the expressions in the infix form that the programmer uses, but this was found to be impractical. There seemed too many tests to be made at several different stages, and bracketed expressions

presented a problem. Therefore the form described earlier was decided on: + and * were to be regarded as n-ary operators; the other three were binary. This was changed later since the occasions when '-' was unary became difficult to handle. It was decided to regard it as unary all the time, treating x-y as x+(-y). A similar change was made for /, treating 1/y as y**(-1) in several routines. The advantage in having to test for three operators instead of five is considerable.

Having decided on the general form, it was necessary to decide how the expression should be stored. The implementing program is a considerable size, and therefore consideration was given to saving space as opposed to time. The English Electric KDF9 and the IBM360/50 were both used in testing the program, and on both space became a problem. The program currently runs on the ICL4-75 under the Edinburgh Multi-Access System, and this does not have the restrictions of the other machines. However the response of AML when being used interactively is reasonably good. (For simple manipulations it is comparable with that of the on-line editor), and at the moment the system causes more delays than AML. Therefore it was decided not to alter the present form of the interpreter. For large jobs, the interpreter will obviously be too slow, and it is envisaged that a compiler may be written for a subset of AML at some later date.

This concern with space lead to the adoption of a linear representation rather than using lists or trees. This meant that most of the adjusting of expressions is done by copying, rather than by adjusting pointers, but as has been said, the penalty does not seem too severe.

There is another reason for this representation. IMP has a type of variable called %string. which seemed ideal for manipulation algebraic expressions. An expression could be stored by an assignment statement,e.g.

S=´+(a,b,*(c,d))´

Strings could be copied from one variable to another by assignments of the form S=T. In addition the concatenation operator could be used to put one expression on the end of another, e.g. S=S.T. Unfortunately strings could not be used in the final program. The maximum length permitted for a string is 255 characters, and this is not enough for general expressions used in AML. However strings were used in the initial development of the algebraic routines, since the programs written with them were much easier to follow. Then when the transformation was made to arrays, the basic structure developed with strings was, of course, preserved.

Thus the algebraic expressions are to be in Polish form and held in byte arrays. There remains the problem of the representation of the basic operands. They could be stored in the form that the user sees them, but this does not convey enough information about their kind. Secondly they could be stored in the form of their analysis record.

However this is too long, and some of the information it gives is redundant at this point. Therefore a compromise notation between the two was chosen, and this was described earlier. This is as concise as it can be while retaining the information which will allow it to be processed easily.

12.1 Processing the (INTOCL).

The algebraic commands of AML are

1) %simplify (INTOCL)?

2) %distrib (INTOCL)?

3) %expand (INTOCL)? (TERM)?

4) %subs (FORCL) %in (INTOCL)?

5) %diff (INTOCL)?

6) %addsum (INTOCL)?

(INTOCL) is a list of names, label expressions and formulae, possibily followed by %into and a label expression list. So for all algebraic commands, the (INTOCL) must be examined first. This is done by a routine called Fillrslt.

Considering lists of length 1, there are six possible forms of (INTOCL).

1) NAME

2) NAME %into LABEL EXPR

3) LABEL EXPR

4) LABEL EXPR %into LABEL EXPR

5) FORMULA

6) FORMULA %into LABEL EXPR.

The results of applying the algebraic command to these different types of arguments is as follows.

1) Puts the result back in NAME.

2) Puts the result in the statement given by LABEL EXPR. NAME is unchanged.

3) Prints out the result, if required, and then loses it. The expression in LABEL EXPR is unchanged.

4) Puts the result in the second LABEL EXPR. The original argument stays in the first LABEL EXPR.

5) Prints out the result, and then loses it.

6) Puts the result in the LABEL EXPR.

In addition to these cases, the (INTOCL) may be ommitted. This is only permissible if the command is not the first of a command list. In this case the result of the previous command is used as the argument.

Since the lists are in general of greater length than one, and since a label expression may refer to a block rather than a single statement, the command may have several arguments. Fillrslt copies these arguments into the array ALG. Thus this is another instance of a use of ALG. The array FORM holds the pointer to the ARN cell.

For a command statement, every argument is copied into ALG, with the array FORM giving the required ARN cell. When all the arguments have been stored, the command is applied to each in turn. If no (INTOCL) is present, the arguments already stored are used. This means that the results of the command must replace the original argument in each case.

Also, up to this point the contents of the original NAME or LABEL EXPR have not been altered.

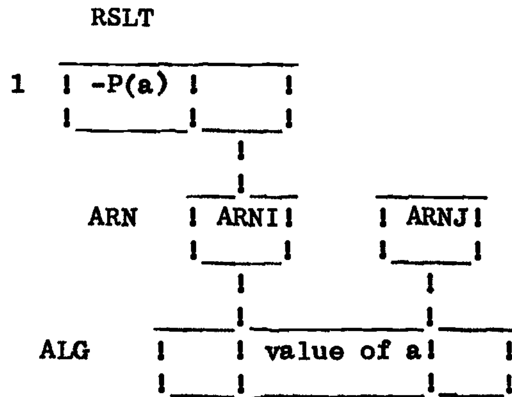
Here is the algorithm for processing the arguments of an algebraic command.

1) Go to (7) if there is no (INTOCL).
2) Fill ALG with the arguments, using FORM to reference them.
3) Get the first argument.
4) Apply the command.
5) Copy the result into ALG, resetting ARNI and ARNJ.
6) If there is another argument, go to (4). Otherwise stop.
7) Fault the program if FORM has no entries. Go to (3).

Associated with the arguments is an array called RSLT. This contains a value showing where the result is to be stored. If the number is positive, it gives the position in the Storage Tree; if it is negative, its modulus is a pointer to INFO. Hence in the first case, the result is stored as a labelled statement, while in the second it is the value of a variable. If the entry in RSLT is 0, the result is not to be stored.

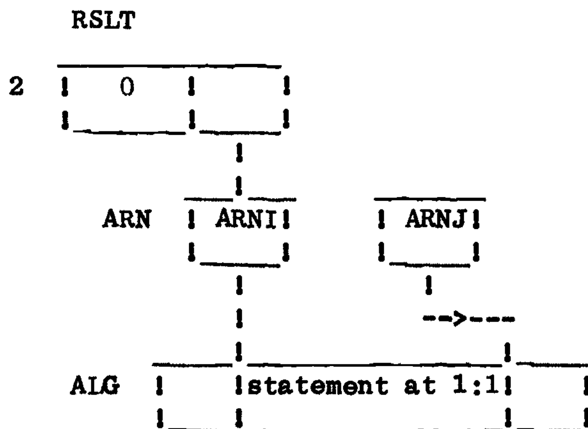
Thus for each argument of the command, Fig. 1 shows the various situations that may arise.

1) e.g. %simplify a

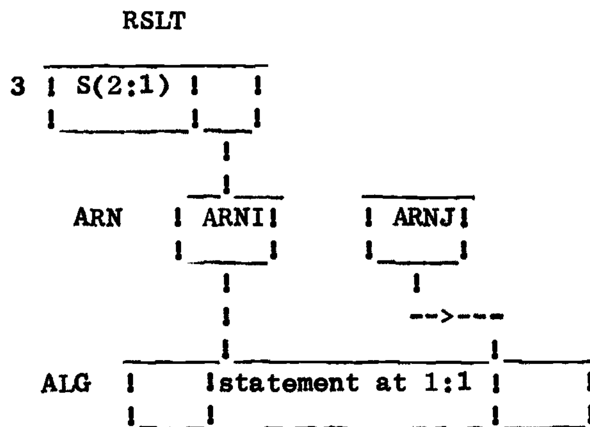


where P(a) is the position in INFO of the description of a.

2) e.g. %simplify 1:1



3) e.g. %simplify 1:1 %into 1:2



where S(2:1) is the position in the Storage Tree of 2:1.

Fig. 1

The following is the algorithm for the routine Fillrslt which enters the arguments into ALG.

1) If there is an '%into', set p pointing to the (LB)* after it. Otherwise p=0. Check that the lists are the same length.

2) Get the first member of the (NAMES) list.

3) If p=0 set q=0, otherwise evaluate the label expression at p into q.

4) If the alternative is NAME, find its position in INFO, and get its value. Call Get to set up ALG, storing the result in FORM. Copy the expression into ALG. If q=0 set RSLT=-j, otherwise set RSLT=q.

5) If the alternative is LB, evaluate its position, k, in the Storage Tree. If there is a statement at k, copy it into ALG. If there is a subtree below k, copy all its statements into ALG, incrementing the pointer to RSLT and FORM before each statement. If q#0, make sure there is a corresponding cell in the subtree below q, inserting one if necessary. (As for the %label command). Set RSLT to this position. Otherwise RSLT=0.

6) For alternative 3, evaluate the first expression into ALG. If (FORMULA) was an equation, evaluate the second expression into ALG, setting the ARN cell pointing to the beginning of the first expression and the end of the second. Set

RSLT=q.

7) Go to (3) if there are any more (NAMES).

The application of a command is done by the routine Manip. This routine decides which command is to be applied, and calls the appropriate routine. When the command has been applied, it puts the result in ALG, destroying the old value, and resetting the ARN cell that FORM points to.

The algebraic commands must each be described seperately. A number of routines, each of which is highly dependent on the others is used. A list of these routines is given with a simple description of their function, and then each one will be described in more detail.

1) COMPARE Compares two expressions to see if they are algebraically equal.

2) ADDEVAL Takes a list of terms separated by '+'. (- is a unary operator more binding than +) and performs any permissible addition. A flag may be set, requesting that the individual terms be simplified before being considered.

3) MULTEVAL Takes a list of terms separated by * signs (a/b having been translated into a*b**-1), and performs any permissible multiplication.

4) EXPN Takes an expression and splits it into base and exponent, changing a**b**c to a and b*c.

5) COLLECT Describes the AML routine Collect.

6) DIFF Differentiate

7) DIST Apply the distributive law

8) EXPD The routine for the command %expand

9) DIV TO MULT Changes all division, x/y, to exponentiation, x*y**-1.

The descriptions of these routines are not accurate. They discuss the algebraic expressions that are handled as if they were strings. In fact, all the expressions are stored in the array ALG, and are described by a base and two relative pointers. The base pointer is a %SHORT %INTEGER %NAME that is set pointing to an ARN cell. The relative pointers give the distance of the beginning and end of the expression from the base.

Example

Suppose one of the routine descriptions refers to the expression P. Then Fig. 2 shows the actual method of storing P.

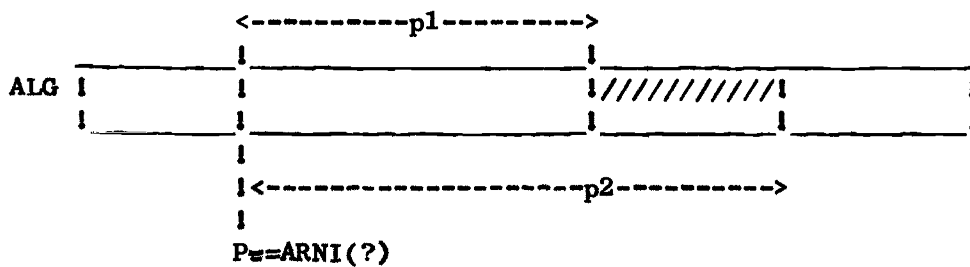


Fig. 2.

The shaded part of ALG is the part actually under consideration.

P = P.X

alters Fig. 2 to Fig. 3, where X was defined in the same way as P.

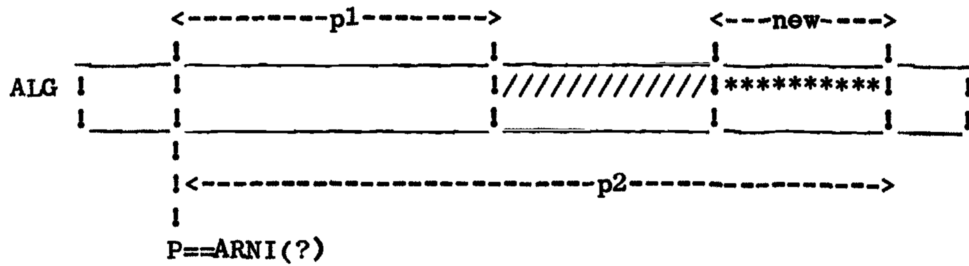


Fig. 3

The part labelled NEW is a copy of the expression X.

12.2 The Routines

1) Skip.

This integer function obtains the next term of the expression in P. The word 'term' in future will mean the part of the expression that Skip yields. This function will be described in more detail than the others.

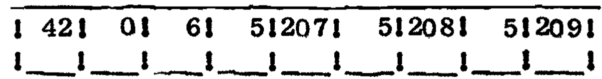
Every expression that is used in the routines used for algebraic manipulation is stored in a byte array in Polish form. For descriptive purposes this is shown in a bracketed form, for example,

*(x,y,z)

for

x*y*z.

Chapter 10 has described how these expressions are built up. Internally the representation of x*y*z would be



where 207, 208, 209 are the hash codes of x, y and z; 42 is the ISO code for *, and 0 and 6 together give the length of

the expression that * is applied to.

Skip has two parameters; P the base pointer of the expression, and an integer p1 which points to the first member of P that is to be considered. The result of Skip is a pointer to the end of the term that is found. Let a be the first member of P. Then the result for Skip is given by the following:

1) If a=* or +, the next two entries of P give the length of the term. So the result is p1+2+L where L is the length of the term.

2) If a=/ or E, the position required is the end of the second operand of a. P1 is advanced by 1, and Skip is called. This gives the end of the first operand. Then Skip is called again to give the end of the second operand.

3) If a=6, then the term is a name with a list, e.g.

6	207	0	0	4	5	208	5	209

for x(y,z).

The second entry after p1 gives the number of primes, and the third and fourth give the length of the subscript list. Hence the result is p1+L+4, where L is the length.

5) If a=5 the term is a name without a list. Hence the result is p1+1.

6) If a=3 or 4, the term is a rational or long real,

which take up nine bytes. So the result is p1+8.

7) Finally if a=1 or 2, the term is an integer or real, so the result is p1+4.

Skip has been described in more detail than the rest of the routines will be. Now that the actual organisation of the algebraic expressions has been described, the routines that handle them will be described in a fairly general manner, using such terms as 'next term of X', 'copy X to Y', etc..

2) Compare(P,Q).

This performs a most fundamental task, as it decides whether or not two expressions are equal. The most general form it has to test can be exemplified by the two expressions

a*b+c*d

d*c+b*a

which it must recognise as being equal. Compare is an integer function which gives the result 1 if the expressions are equal, and 0 otherwise. The two expressions would be presented to Compare in Polish form, i.e. as

+(*(a,b),*(c,d))

+(*(d,c),*(b,a))

Compare is simply recursive. It assumes that it may receive three kinds of expressions.i.e.

1) Those beginning with * or +

2) Those beginning with / or E.

3) Those consisting of a single operand.

In the first two cases, the result is 0 if the two operators are not equal.

Case 3.

If the single operand is a name or constant, then the two expressions can be compared exactly. The result is 1 if they are identical, and 0 otherwise.

If they both consist of the same name followed by a list, then each element of the list must be equal to its partner. For example

a(x+y,z)

and a(y+x,z)

are equal, but

a(x+y,z)

and a(z,x+y)

are not.

The number of entries in each list must be the same. Compare is called for each set of pairs, and the result is 1 only if all the pairs are equal.

Case 2.

Since the two operators are non-commutative, the first operands of the expression must be equal, and so must the second two. Compare is called for each pair of operands.

Case 1

This time, since + and * are n-ary operations, each operand in one expression must be checked against every operand of the other. Let use consider an example using *,

e.g. comparing

P = *(a,b,c)

and Q = *(c,a,b).

This is done in the following way.

1) Let S be the first term of P.

2) Let T be the first term of Q.

3) Compare S and T, and go to (6) if they are equal.

4) If T was the last term of Q, nothing has been found to equal S. Therefore the result is 0.

5) Get the next term, T, of Q and go to (3).

6) Since S and T are equal, we now want to compare the remaining terms of P and Q. In the example, ´a,b,c´ is compared with ´c,a,b´. Once the a´s have been matched, ´b,c´ is compared with ´c,b´. If Q=T and S is the last term of P, the result is 1, since all terms have been matched. If Q=T, but P contains a term after S, the result is 0. Otherwise, resolve Q -> Q1.(T).Q2, and then set Q=Q1.Q2, thus eliminating T. Get the next term, S, of P, and go to (2). The result is also zero if there is not another term of P.

3) Addeval(P,R,mult)

An expression in Polish notation is given as the parameter P. The routine examines each term of the expression, to add together those which differ by a constant. For example,

+(*(2,a,b,c),*(3,b,c,a))

will give the result

*(5,a,b,c)

Constants are also added together, in an array called FACT. The routine calls a local function called Split P. This has as parameters arrays F and K. The next term of P is removed. If it is a number it is added to FACT, and the result is zero. Otherwise the constant in the term is put into K, and the rest of the term is put in F. The result is one. For instance, in the example given above, the first term was *(2,a,b,c). Split P divides this so that the 2 goes in K and a,b,c goes in F, returning the result 1.

Having said this, the algorithm for addeval can be given.

1) Set FACT=0

2) Call Split P to get the first term of P, putting the constant in CONST, and the rest in F.

3) Go to (8) if Split P gave the result 0.

4) Go to (8) if there are no more terms in P.

5) Call Split P again, putting the result in C and X. Go to (7) if Split P is 0.

6) If Compare (X,F)=1, the constants that multiply them may be added together. The result of this is put back in CONST. Otherwise the term C*X is saved in Q.

7) Go back to (5) for the next term of P, if there is one.

8) The value CONST*F is put in R, unless CONST=0.

9) Q has all the terms that did not agree with F. Therefore if Q is non-empty, copy it into P, and go to (2).

10) The result is FACT + R.

Example

+(*(3,a,b),*(2,b,a),b,a,b,2,*(3,c),5,*(4,c))

becomes +(7,*(5,a,b),*(2,b),a,*(7,c))

I.e. 3*a*b + 2*b*a + b + a + b + 2 + 3*c + 5 + 7*c

becomes 7 + 5*a*b + 2*b + a + 7*c.

Additional tests, not described here, must be made for negative terms. If the parameter mult is set to 1, the term found by Split P has Multeval applied to it, before being split into

constant * rest,

and being copied into F and K. After each term of P has been processed once, mult is set to zero, to prevent unnecessary applications of Multeval.

4) Expn(P,p1,ptr1,ptr2,R)

This is an integer function which evaluates the exponent of a term. For example, EEEabcd is evaluated to

a to the power *(b,c,d).

The function works on the Polish expression which is held in P, beginning at the position p1. The base is unaltered by this function; it is merely found, and the pointers ptr1 and ptr2 are set to the beginning and end of it. The exponent is

copied into R. If the expression evaluates to a numeral, this is put in R and the result of the function is 0. Otherwise the result is 1. On exit, pl points to the first position in P after the exponent.

Here is the algorithm for the function.

1) Count the number of E's, putting the answer in i.

2) Set ptrl and ptr2 to the beginning and end of the base. Set pl pointing immediately after the base.

3) If i=0, set R empty and go to (8).

4) Do (5) i times.

5) Get the next term of P, calling Addeval if it begins with +, and copy the result into Q.

6) Call Div-to-mult and Multeval for Q, putting the result in R.

7) Result=1 unless R is numeric.

8) Go to (10) unless the base is numeric.

9) Let B be the base. If i=0, R=B, otherwise R=B**R. Result=0.

10) If R=0, set R=1. Result is 0.

11) Set R empty if R=1. Result=1.

5) Multeval(P,R)

This routine has in its parameter P a Polish expression of terms multiplied together. If any terms have the same base, the exponents are added, e.g.

*(Ea2,b,Eac,Eb3)

becomes *(Ea+(2,c),Eb4).

The result is put into X.

Expn is applied repeatedly to P, thus obtaining pointers to the base of each term. The exponent is held in R. In fact the same R is used each time, pointers being used to delimit the exponents. When all of P has been processed, we have the following situation.

All bases are held in P, with pointers to the beginning and end of each base. All exponents are held in R, with pointers to the beginning and end of the exponents. Let P(i) be the i-th base of the expression, and R(i) be the i-th exponent.

If any of the terms was found to be numeric, it is used to multiply the constant factor 'fact', which initially is one. N holds the number of terms of the form P(i)**R(i) that were found in P. If n is 0 when P has been processed, the result is given by fact. The algorithm following describes the case for n>0.

1) Set i=1.
2) Go to (11) if P(i)=''. (Already dealt with, see (8).)
3) If R(i) is empty, set Y=1; otherwise set Y=R(i).
4) Go to (10) if i=n (I.e. P(i) is the last term of P).
5) Set j=i+1.
6) Go to (9) if P(j)=''.
7) Compare P(i) and P(j). Go to (9) if the result is 0.

8) Copy R(j) into Y (or copy 1 if R(j) is empty). Set P(j)='' to indicate that the term has been used.

9) Increase j by 1. Go to (6) unless j>n.

10) Call Addeval for Y. If the result is 1, copy P(i) into X. Otherwise copy P(i)**Y into X, provided Y is not zero.

11) Increment i. Go to (2) unless i>n.

12) Copy fact in front of X. Result is in X.

8) Dist(P,X)

This routine applies the distributive law to the expression in P, putting the result in X. Examples of the transformation required are

```
*(+(a,b),c)  ->  +(*(a,c),*(b,c))
E*(a,b)c  ->  *(Eac,Ebc)
*(+(a,b),c,E*(p,q)-1,Er-1,E*(s,t)-1) ->
   *(+(*(a,c,Er-1),*(b,c,Er-1)),
      E+(*(p,s),*(q,s),*(p,t),*(q,t))-1)
```

```
(I.e.  (a+b)*c*(p+q)**-1*r**-1*(s+t)**-1 =
       a*c*r**-1+b*c*r**-1*(p*s+q*s+p*t+q*t)**-1 )
```

Div-to-mult is assumed to have been applied to P before it is passed to Dist.

First let use consider an expression with no exponential expressions of the form E*(a,b)c, and with no denominator. For example, let us consider the expression

```
*(a,+(b,c,d),e,+(f,g),h,+(j,k))
```

This is broken down in the following manner

1) a,e and h, i.e. the terms that do not have + in them, are copied into R.

2) The sub expressions containing + are broken up into terms, so that P(i,j) is the j-th term of the i-th subexpression of P. The following table illustrates this.

P(1,1) = b

P(1,2) = c

P(1,3) = d

END(1) = 3

P(2,1) = f

P(2,2) = g

END(2) = 2

P(3,1) = j

P(3,2) = k

END(3) = 2

m = 3

This table is used by a recursive routine local to Dist, called FillX(i,j). The algorithm for it is as follows.

1) Set r2=r1, pointing to the next free space in R.

2) Call Dist to put P(i,j) into R.

3) If i=m do (4); otherwise call FillX(i+1,1).

4) Call Multeval to copy R into X.

5) Set r1=r2, going back to the position we started with.

6) Call FillX(i,j+1) unless j=END(i).

7) Set r1=r2. Return.

R1 is a pointer global to FillX. R2 is local. FillX(1,1) will perform the required distribution. Let use examine the working of FillX, using the example given above. Initially R='a,e,h' and X is empty

I Call FillX(1,1)

2) This copies P(1,1) into R, so that

R = 'a,e,h,b'

3) i#m, so

II Call FillX(2,1)

2) This sets R = 'a,e,h,b,f'

3) i#m, so

III Call FillX(3,1)

2) R = 'a,e,h,b,f,j'

3) i=m so copy R to X, i.e.

4) X = '*(a,e,h,b,f,j)'

5) Reset r1, so R = 'a,e,h,b,f'

6) j#END(i), so

IV Call FillX(3,2)

2) R = 'a,e,h,b,k'

3) i=m so copy R to X, i.e.

4) X ='*(a,e,h,b,f,j),*(a,e,h,b,f,k)'

5) Reset r1, so R = 'a,e,h,b,f'

7) j=END(i), so return to

III, step (6)

7) j=END(i), so return to

II, step (3)

5) Reset r1, so R = 'a,e,h,b'

6) j#END(i), so

V Call FillX(2,2)

2) R = ´a,e,h,b,g´

3) i#m, so call FillX(3,1)

This adds two more terms to X, as before, but with g instead of f. So X is ´*(a,e,h,b,f,j),*(a,e,h,b,f,k),*(a,e,h,b,g,j), *(a,e,h,b,g,k)´

5) Set r1=r2, so R =´a,e,h,b´

6) j=END(i), so return to

II, step (6)

7) Return to

I, step (3)

5) Reset r1, so R = ´a,e,h´

6) j#END(i), so

VI Call FillX(1,2)

2) R = ´a,e,h,c´

3) i#m, so call FillX(2,1)

This adds four more terms to X, as before, but with c instead of b. So X is now ´*(a,e,h,b,f,j),*(a,e,h,b,f,k),*(a,e,h,b,g,j), *(a,e,h,b,g,k),*(a,e,h,c,f,j),*(a,e,h,c,f,k), *(a,e,h,c,g,j),*(a,e,h,c,g,k)´

5) Set r1=r2, so R =´a,e,h,c´

6) j#END(i), so

VII Call FillX(1,3)

2) R = ´a,e,h,d´

3) i#m so call FillX(2,1)

This adds four more terms, with d replacing b this time. So X is

´*(a,e,h,b,f,j),*(a,e,h,b,f,k),*(a,e,h,b,g,j),
(a,e,h,b,g,k),(a,e,h,c,f,j),*(a,e,h,c,f,k),
(a,e,h,c,g,j),(a,e,h,c,g,k),*(a,e,h,d,f,j),
(a,e,h,d,f,k),(a,e,h,d,g,j),*(a,e,h,d,g,k)´

5) Set r1=r2, so R= ´a,e,h´

6) j=END(i), so return to

V, step (6)

7) return to

I, step (6)

7) Return to Dist.

Now let us suppose that Dist is applied to an expression which only contains products raised to a power, for example

P = EEE*(a,b,c)x y z

In this case, Expn is applied to P, putting the exponents into Q. Then if the base begins with a ´*´, each of its terms is raised to the power given by Q. Otherwise E▮bQ is copied into R, where ▮b is the base. (This cannot be numeric, see Expn.)

We may now consider Dist applied to a general expression P. It divides P into three parts:

(i) terms involving + which are not raised to a power

(ii) terms involving + raised to the power -1.

(iii) terms not involving +, or ones which are raised to a power other that -1.

P is assumed to be a series of terms multiplied together. Dist processes the expression P by the following steps

1) Let X be the first term of P

2) Go to (5) unless X is of the form +(A1,A2,...An).

3) Split up X, setting G(i,j)=Aj, where i is the number of terms of this form that have so far been found in P.

4) If X is not the last term of P, get the next term, call it X and go to (2). Otherwise go to (8).

5) Call Expn(X,Q), which leaves the base of the term in Xb, and puts the exponent, if any, in Q.

6) Go to (7) unless Q=-1 and Xb is of the form +(B1,B2,...Bn). Split up Xb, setting H(k,j)=Bj, where k is the number of terms of this form found so far in P. Go to (4).

7) Copy the term into R, raising each term of Xb to the power Q, if Xb begins with a *. Go to (4).

8) If i=0, Multeval R into X. Otherwise call FillX(1,1) for G.

9) If k>0 call FillX for H, putting **-1 after the result.

10) Return.

9) Diff(F,R,dx)

This routine differentiates the expression in F with respect to the algebraic constant whose hash code is in dx. The result is put in R.

The routine uses a local integer function Hasdx(Y) which examines an operand and determines whether it contains the name given by dx. Its results can be tabulated as follows.

1) If the operand is a constant or a name other than dx, the derivative is zero. Hasdx returns the result 0.

2) If the operand is dx, the derivative is one. Hasdx returns the result 1.

3) If the operand is the name dx followed by a list, this is treated as dx, so the result is 1.

4) If the operand is a name other than dx, followed by a list, there are two alternatives

   i) The name may be a trigonometric name that has a recognised derivative. In this case the derivative is copied into the array Y.

   ii) If the name is not recognised, a prime is added after the original name (So that f(x) becomes f′(x), f′(x) becomes f′′(x), etc.).

   The arguments of the name are differentiated using Diff. The derivatives are copied into Y. If the derivative of the argument is 1, the result returned by Hasdx is 2; otherwise it is 3.

5) If the operand is an expression, it is differentiated into Y and Hasdx returns the value 4.

If the expression presented to Diff is a sum of terms in dx, each term can be considered individually, since

$$(f+g)' = f' + g'.$$

However if the expression contains a product of terms in dx, the result is more complicated.

$$(f*g)' = f'*g + g'*f.$$

Hence for each term that is differentiated, both the derivative an the original term must be kept, in case the term is part of a product. If the expression is a product of three terms, the same rule as above can be applied. E.g.

$$(f*g*h)' = (f*g)'*h + (f*g)*h'$$

which is $f'*g*h +g'*f*h + f*g*h'$.

The algorithm for dealing with products in Diff is

1) Get the first term. Set Z and X empty.
2) Put the term in F, and its derivative in Y.
3) Copy Y into X if Y is non-empty.
4) Go to (9) if Z is non-empty.
5) Go to (7) if this is not the last factor of the product.
6) If X is empty, the P=1, else P-X. Finished.
7) If X is empty, then Z=1, else Z=X.
8) Copy F to the end of G. Get next term. Go to (2).
9) Set Z=F*Z+X*G. Go to (8) if there is another factor. Copy Z into P.

Let us consider this algorithm using the product f*g*h.

1) Z = '', X = ''

2) F = f, Y = f'

3) X = f'

7) Z = f'

8) G = f

2) F = g, Y = g'

3) X = g'

9) Z = g*f' + g'*f, X = g'*f

8) G = f*g

2) F = h, Y = h'

3) X = h'

9) Z = h*g*f' + h*g'*f + f*g*h'

R = h*g*f' + h*g'*f + f*g*h'

The individual terms of a product may contain an exponential sign (/ is replaced by **-1) which must be tested for. The two possible cases are

f(x)**c

and c**f(x)

which give derivatives

c*f(x)**(c-1)*f'(x)

and f'(x)*c**f(x)*log(c).

F(x)**g(x) is faulted. In addition, there may be constant terms, which must be saved in case they are needed in a product.

The expression F is differentiated with respect to x by the following algorithm.

1) F is assumed to be a number of terms added together. Let P be the first of these.

2) P is therefore a number of terms multiplied together. Let Q be the first of these.

3) Go to (6) unless Q is of the form ′E′.S.T.

4) Apply Hasdx to S and T, to see whether either of them is dependent on x. Fault the program if they are both functions of x.

5) If S=S(x), let Y have the derivative of S. Set G=T*S**(T-1). Otherwise, if T=T(x), let Y have the derivative of T. Set G=Q*log(S).

6) If Q is not of the form ′E′.S.T, apply Hasdx to Q. Let Y have the derivative of Q if Q=Q(x). Go to (12) if Q is independent of x. (whether or not it begins with ′E′).

7) Set G=G*Y. (Assuming that G is 1 if Q is not ′E′.S.T.)

8) If Q is the only term of P to be dependent on x, the derivative of P is Pr*G, where Pr represents all terms of P other than Q. In particular, if Q is the last term of P, and if Z is empty (see below), this is so. Set R=R+Pr*G. R holds the result, and is initially set to zero. This gives the result if P is the last term of F. Otherwise, let P be the next term of F and go to (2).

9) Assume for a moment that Q is the first term of P to be dependent on x. However, Q is not the last

term of P. In this case we must save the value of G, and the value of Q, in case there is another term dependent on x. Set Z=G and V=Q.

10) If Q is not the first term of P dependent on x, Z will be non-empty. Therefore we have a product of terms:

Q whose derivative is G

and V whose derivative is Z.

The derivative of the product is therefore

Q*Z+V*G

and the product itself is Q*V.

If Q is not the last term of P, we wish to save these values, in case there is another term to be considered. Therefore Z=Q*Z+V*G and V=Q*V. Then the next term, Q, of P is considered by returning to step (3).

11) If Q was the last term of P, the derivative of P is Pr*Z, where Pr contains all the terms of P that are independent of x. Therefore set R=R+Pr*Z, and process the next term, P, of F, by returning to step (2).

12) The only case left to consider is that where Q is independent of x. If Q is the last term of P, and if Z is empty, the derivative of P is zero, Therefore process the next term, P, of F immediately.

13) Q is saved in Pr. If it is the last term of P,

Pr*Z is added to R, and the next term of F is processed. Otherwise the next term of P is processed.

10) Div-to-mult(P,S)

This takes an expression P which may have division signs in it, and returns the expression S, in which all division signs have been converted.

Consider, as an example, the expression

*(///ab+(c,/de)f,Eg/bc,/a/b/cd)

which is the Polish notation for

$$\left(\frac{a/b}{c+d/e}\right) / f*g^{(b/c)} * \left(\frac{a}{b/(c/d)}\right)$$

This is changed to

$$ab^{-1} * (c+de^{-1})^{-1} * f^{-1} * g**(bc^{-1}) * ab^{-1}cd^{-1}$$

Div-to-mult processes the expressions as follows

1) Count the number of / signs at the beginning of the expression.

2) Let P be the first term after the /s.

3) Go to (5) unless the number of /s is zero.

4) If P begins with +, *, or E, call Div-to-mult for each of its terms, putting the result in R. Otherwise copy P to R.

5) Return if P is the last term of S. Otherwise discard P, and go to (1).

6) If there were n / signs, do (7) to (9) n times.

7) Get the next term, and call Div-to-mult for it, putting the result in Q. Q is therefore a series of terms multiplied together.

8) Let T be the first term of Q.

9) If T is of the form X**-1, put X in R. Otherwise put T**-1 in R. Go to (7) if T is the last term of Q. Otherwise let T be the next term, and go to (8).

10) Go to (5).

11) Expd(G,R,n,m)

The expression G**n is to be expanded, putting the result in R. If m is zero, all terms are to be found. Otherwise the m-th term is required. N must be an integer or rational number. Terms is a global integer which indicates the number of terms required. Normally it is 5, but it may be reset by the programmer. The algorithm for Expd is as follows.

1) Set fact=n. Go to (3) unless n=1.

2) If m=0 or 1, set R=G. Otherwise R=0. Return.

3) G is a series of terms added together. Set G1=first term of G, G2=rest of G.

4) Go to (6) unless G1=1. Go to (2) if G2 is empty. Copy G2 to P. Set Q and F empty.

5) Set R=1 if m<2. Go to (9) if n is a positive integer. Set i=terms. Go to (11).

6) Set Q=G1**n. Go to (7) unless G2 is empty. Call

Dist(Q,R) and then return.

7) Go to (10) unless n is a positive integer. Set P=G2, F=g1.

8) If m<2 call Dist(Q,R). Empty Q.

9) If n>terms then i=terms else i=n.

10) Set R=1 unless m>2. Set P G1**-1. F is empty. Set i=terms.

11) Return if m=1.

12) Do (13) to (17) for k=1,1,i.

13) Go to (17) unless m=0 or k=m-1.

14) Set A=F**(i-k).

15) Call Expd(P,A,k,0). Set A=fact*A.

16) Call Dist(A,R). Go to (18) unless m=0.

17) Set fact=(n-k)/(k+1)*fact.

18) Return if Q is empty. Dist Q*R into R and return.

12) Subst(P,Q,R,A)

This routine substitutes the expression in P for each occurrence of Q that is found in R, putting the result in A. The actual substitution is done by a local integerfn Sub(R,A). In fact, Subst merely sets up some local variables which must be global to the recursive function Sub. Sub is a function which returns the value 1 if some substitution was done, and 0 otherwise. It has two parameters, R and A, since P and Q will remain the same for all calls of Sub.

Seven cases may arise in this routine, and they are listed below. Sub tests for these cases in the order they

are given.

1) Q=R.

In this case, P is copied into A, and the result of Sub is 1.

2) Q and R begin with the same operator, which is + or *.

For example,

Q=+(q1,q2,...qn)

and

R=+(r1,r2,...rm)

The routine tests for the case where for some i,

Ri+k = Qk, for k=1,2,...n

E.g.

R=+(a,b,c,d,e)

and

Q=+(c,d).

This test is done by two local routines. The first, Firstmatch searches down R, until it comes to an Rk which is equal to Q1. This equality is found by a local integer function, Eq(R,Q) which gives the value 1 if they are equal, and 0 if they are not. If Firstmatch is sucessful, the second function, Compare-rest is called. This moves down Q and R, from Q1 and Rk, testing that the successive pairs are equal. (Q must have at least two terms to justify the use of the operator.) If Eq fails for one of the pairs, or if there are less than n-1

Rj's after Rk, Compare-rest fails. It succeeds if it manages to match all the Q's to terms of R. In this case, Rk,Rk+1,...Rk+n is replaced by P. Before discussing this case further, however, we must consider the case when Firstmatch is unsucessful. It is still possible to perform some substitution, for it is possible that the whole of Q is contained in one of the Rj's. For example,

Q=+(a,b)

and R=+(*(c,+(a,b)),d).

Hence Sub(Rj,A) is called for each term, Rj of R. This process must also be carried out for the terms up to Ri, if Firstmatch is a success. Hence the steps taken are

i) Call Firstmatch, letting i point to the term that matched Q1. If there is no such term, set i=m+1.

ii) Call Sub(Rj,A) for j=1,1,i-1

iii) Process is finished if i=m+1.

iv) Call Comparerest.

v) If this is successful, copy P to the end of A. If i+n-1=m, the process is finished. Otherwise consider Ri+n as R1 and go back to (i).

vi) If Comparerest fails, it is possible that a match can still be found. For example,

R=+(a,a,b,c)

Q=+(a,b)

will fail for the first a, but succeed for the second. Therefore call Sub(Ri,A) and return to (i), treating Ri+1 as R1.

3) R begins with * or +, but Q has a different operator, if any.

Sub is called for each term of R.

4) R is the AML function 'power', e.g.

R = power(E,X,V,I,J).

which means the sum for R I,I+1,... J of

E*X**V

where E may be a function of V.

It is this situation which causes the complications in Subst. First Sub(E,T) is called. If this returns a value 1, T replaces E in R, the resulting expression is copied into A, and the process is finished. If the value 0 is returned, we consider another possibility. Suppose for example, that

Q = 3*a

and R = power(r*a,x,r,0,10).

Then the result obtained is

A = y*x**3+power(r*a,x,r,0,2)+power(r*a,x,r,3,10),

assuming that P=y. The method of doing this is as follows. The variable sigflag, which is global to Sub is set to 1. The arrays MF and NF, also global to Sub, are given the values I and J. Now we can discuss the function Eq.

If sigflag=0, the result is 1 only if Q and R are identical. If sigflag is 1, however, another possibility is allowed. The result is 1 if R is the variable V, and Q is a number lying between MF and NF. If this is so, Q is copied into the array NO, and if a subsequent value of Q is found, satisfying these conditions, it must be equal to NO. This ensures that for example,

Q = 3*a+3*a**2

will be accepted for

R = power(r*a+r*a**2,x,r,0,10),

but

Q= 2*a+3*a**2

will not. Once sigflag has been set to 1, thus allowing this additional form of 'equality', Sub(E,T) is called again. If it still gives the result 0, then R is copied into A, and the result of Sub(R,A) is 0. Otherwise A is constructed in the following manner.

T is set to T*X**V, and Subst(NO,V,T,A) is called, so that the number found replaces all instances of V in the expression. Then

power(E,X,V,MF,NO-1) + power(E,X,V,NO+1,NF)

is added to the end of A.

13) Addinf(S,G)

This routine searches S for any instances of the AML function 'power'. If it finds more than one, it attempts to add them. Thus for example, if

S = power(a,x,r,1,10) + power(b,x,r,5,20)

the result, G, is

power(a,x,r,1,4) + power(b,x,r,11,20)

+ power(a+b,x,r,5,10)

To illustrate the procedure of Addinf, we shall consider a more complicated example of S, i.e.

power(a,x,r,1,50) +power(b,x,r,10,30)

+ power(c,x,r,10,70) + power(d,x,r,70,INF)

Any terms of S that are not the function power are copied straight into G. The remaining terms are split into parts as illustrated by the table in Fig. 4.

	A	X	R	M	N
1	a	x	r	1	50
2	b	x	r	10	30
3	c	x	r	10	70
4	d	x	r	70	INF

Fig. 4

The following is an algorithm for the routine Addinf.

1) Set k = 1, i = no of entries in the table.

2) Set r = i.

3) Do (4) for j=k+1 to r in steps of 1.

4) Compare the entry j in the table with the entry k, putting any new function found at the end of the

table, increasing i if there is a new function, and altering the existing entries at j and k.

5) Increase k by 1, Go to (2) if $k<i$.

This process could apparently go on indefinitely, but because of the nature of the process, the possibility of getting a new term decreases. This will be discussed further when the process has been examined in detail. Step (4) must now be broken down. It is concerned with examining entry j and entry k, and providing a new result if there is one. This is done in the following way.

1) No action is taken unless X(j) = X(k), since the functions cannot be added if the power series variables are different.

2) If R(j)#R(k), this makes no difference, since the choice of summation variables is arbitary. R(j) replaces R(k) in A(k).

3) For the moment we may ignore the expesssion, and consider only the bounds. Addition can only take place if

   N(j)>=M(k) and N(k)>=M(j)   (i)

   This can be seen by considering them as intervals on the real line. Let us fix the pair (M(j),N(j)).

```
--------[---------]---------
       M(j)      N(j)
```

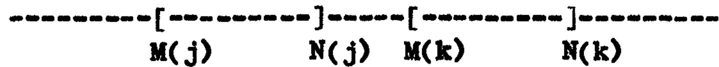
   We already have the relations

   $M(j)<N(j)$ and $M(k)<N(k)$

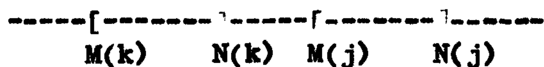
Suppose now that $N(j)<M(k)$. This fixes the relation of the four variables, i.e.

$$M(j)<N(j)<M(k)<N(k),$$

and so we have two non-intersecting intervals, which cannot be combined.



Similarly, if $M(j)<N(k)$, we have the two intervals arranged as follows



4) If condition (i) is satisfied, there will be a new term. Hence A(j) and A(k) are added.The result is put in the next entry of A in the table, and i is increased. Ai points to this new entry.

5) The remaining calculation is concerned with filling in the bounds for the new term, and adjusting those of j and k. (X(j) and R(j) are copied into X(ai) and R(ai))

6) Two flags, lm and ln are set so that

$$ln = 2 \text{ if } N(j) = N(k)$$

$$ln = 1 \text{ if } N(j) < N(k)$$

$$ln = 0 \text{ if } N(j) > N(k),$$

and similarly for lm. First consider the case where lm=2 and ln=2, i.e.

$$N(j) = N(k) \text{ and } M(j) = M(k).$$

Here we require the bounds of ai to be the same as those of j and k, and the entries j and k are no

longer needed. Hence N(ai) is set to N(j), and M(ai) is set to M(j). A(j) and A(k) are set to 0 to indicate that the terms are no longer required.

The remaining cases have a symmetry about them. In order not to lose this, a routine called Test is used. Fig. 5 tabulates the values of lm and ln, the result required, and the call of Test.

	lm	ln	Mj	Nj	Mk	Nk	Mai	Nai	Test
1	2	1	-	-	Nj+1	Nk	Mj	Nj	(2,j,k)
2	2	0	Nk+1	Nj	-	-	Mj	Nk	(2,k,j)
3	1	2	Mj	Mk-1	-	-	Mk	Nj	(3,j,k)
4	1	1	Mj	Mk-1	Nj+1	Nk	Mk	Nj	(0,j,k)
5	1	0	-	-	Mk & Nj+1	Mj-1 & Nk	Mj	Nj	(1,j,k)
6	0	2	-	-	Mk	Mj-1	Mj	Nj	(3,k,j)
7	0	1	Mj & Nk+1	Mk-1 & Nk	-	-	Mk	Nk	(1,k,j)
8	0	0	Nk+1	Nj	Mk	Mj-1	Mj	Nk	(0,k,j)

Fig. 5

It will be noticed that on two occasions an extra instance of the entry of either j or k must be made. An example of the situation in line 5 is

power(a,x,r,5,10) + power(b,x,r,1,20)

which gives the result

power(a+b,x,r,5,10) + power(b,x,r,1,4)

+ power(b,x,r,11,20)

The algorithm for Test will be given. To avoid

confusion, different formal parameters for Test have been chosen:

Test(v,p,q)

1) Set Y=M(p), M(ai)=Y.

2) Go to (10) if v=1.

3) Set C=N(q),N(ai)=Q.

4) Go to (8) if v=2.

5) Set N(p)=Y-1.

6) Go to (8) if v=0.

7) Empty A(q). Return.

8) M(q)=c+1.

9) Empty A(p) unless v=[illegible]. Return.

10) Copy new instance of p into table (entry i).
    Set N(i)=N(p), C=N(q), N(ai)=c.

11) Set M(i)=c+1. Go to (5).

In addition, every time an entry in the table is altered, a test is made to see if M=N for that entry. If this so, the function

power(a(r),x,r,m,m)

is replaced by

a(m)*x**m.

This expression is copied into G, and the entry in the table is erased. Finally the entries are reconstructed into 'power' functions, and copied into G.

The Termination of Addinf

Let there be n entries in the table initially. Set k=1, p=n, and j=2. Assume that j and k overlap. The the two overlapping entries are replaced by (at most) three discrete entries.

Increase j by 1, and again assume that they overlap. These are replaced by three discrete entries. Thus, after performing the cycle once, we have at most (n-1) additions. None of these entries overlap with the first, and the rest may be divided into discrete pairs. This means that when k is set to 2, and the cycle is repeated, there is at least one entry which does not overlap k. Hence if we set m=2(n-1), the maximum number of additions possible is m-2.

Thus each time the cycle is re-started, the maximum number of additions to the table is decreased by one, which means that ultimately the number of possible additions is zero, and so the process will stop.

XIII Implementing AML - Patterns

Two of the conditions allowed in AML use patterns. They are

(NAME)(LIST)? '%matches' (PATTEXPR)

(NAME)(LIST)? '%contains' (PATTEXPR)

Since patterns may contain extractors which point to particular parts of a expression, the expression being examined must be stored. Therefore the location of an expression is given in the condition. A name, and not the label of a statement must be used because of the internal representation; the statement at a label is held in infix form, but the value of a variable is held in Polish form. The restrictions are not necessary if there are no extractors in the pattern, but it was felt to be less confusing if only one form is permitted for all pattern matching.

Let us consider an example.

```
        x = y + z + q
1:1     a <- p*r
 :2     write(x)
        %do 1 %if x matches a_%algebraic +b_%any.
```

The value of x is held in Polish notation in ALG. i.e.

+(y,z,q)

This expression is obtained from the ARN cell that INFO(i+1) points to, where i=VAL(j), and j is the hash code of x. Fig. 1 shows the situation.

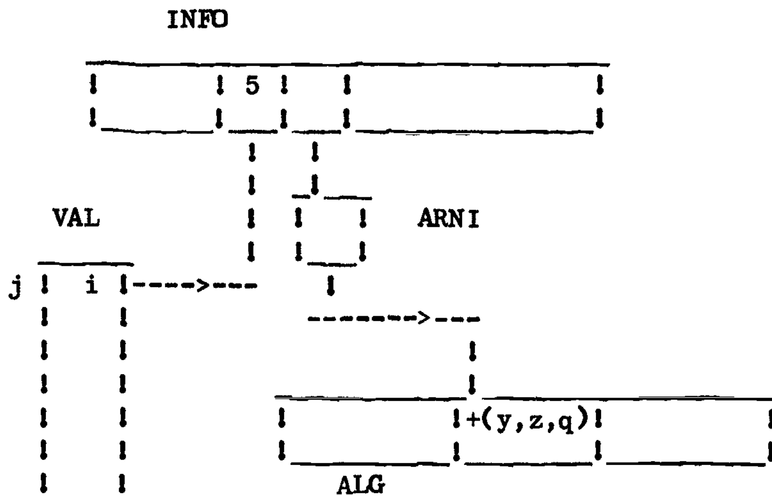


Fig. 1

The position INFO(i+2) is used to point to the first extractor, a. If k is the hash code of a, and m=VAL(k), the five locations INFO(m) to INFO(m+4) are used as follows.

INFO(m)	Set to 15 to indicate an extractor.
INFO(m+1)	point to the beginning and end of the
&	part of the expresssion that a accesses.
INFO(m+2)	They are set relative to the beginning of the expression. Hence, in this example, x is stored in ALG as

+	0	6	5	Hy	5	Hz	5	Hq	1	0	0	0	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13

where Hy, Hz and Hq are the hash codes of y, z and q respectively.

INFO(m+1)	is 3
INFO(m+2)	is 4.
INFO(m+3)	contains the operator ′+′, and also the hash code of b. The use of ′+′ will be

discussed later. For the extractor b, INFO(p+3) contains J as well as ´+´, since b is the last extractor pointing to x. (P being the position in INFO where b´s description begins.)

INFO(m+4) points back to the position of x´s description in INFO.

Fig. 2 shows the situation at this stage.

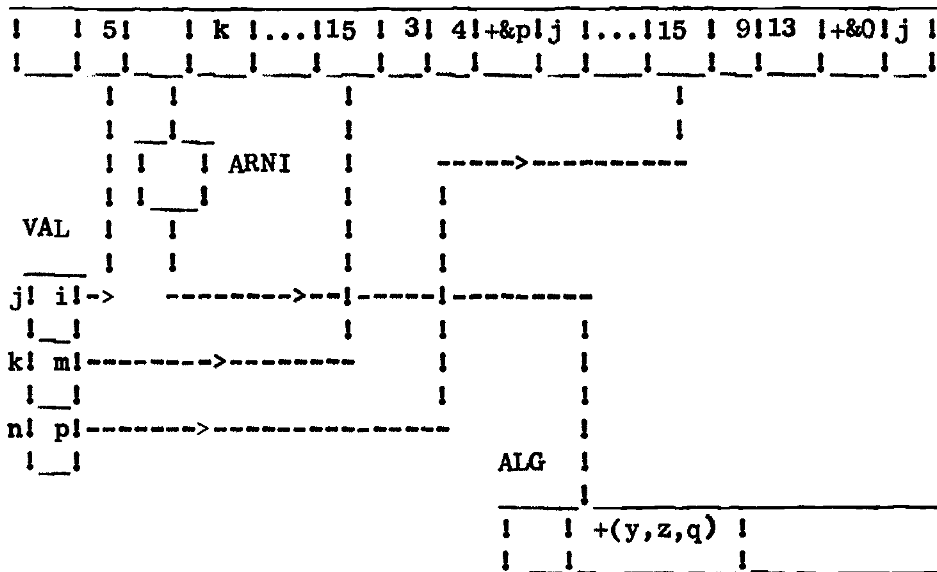


Fig. 2

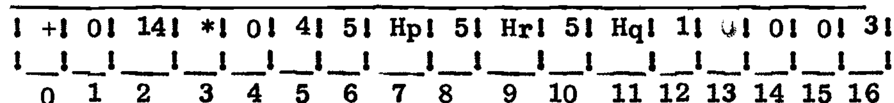
The arrow in statement 1:1 means that p*r should replace the value that a points to in x. An ´=´ sign, e.g. a=p*r, would remove the pointers to x, and treat a as an ordinary variable. However, with the left arrow, three things are necessary.

1) The expression ´p*r´ must replace y in x, and the new expression must be stored in ALG.

2) The pointers of a must be changed.

3) The pointers of the other extractors that reference x must be changed.

A routine called Replace is used to substitute p*r for y. This creates a new version of x, and resets the values of INFO(m+1) and INFO(m+2), using name type parameters. This new version is put in ALG, deleting the old version, and resetting the ARN cell. Now the pointers of the other extractors for the expression must be changed. This is done by using the pointer to the first extractor that is in INFO(i+2), and then the pointer to INFO(m+3), etc., to access the remainder. If any of the extractors point to a part of x after the altered one, as b does, the value length, given by Replace, is added to INFO(p+1) and INFO(p+2). Length, which may be positive or negative, is the difference in length between the old and new values of a. X is now.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+	0	14	*	0	4	5	Hp	5	Hr	5	Hq	1	0	0	0	3

So length = 3.

Fig. 3 shows the situation after the alterations have been made.

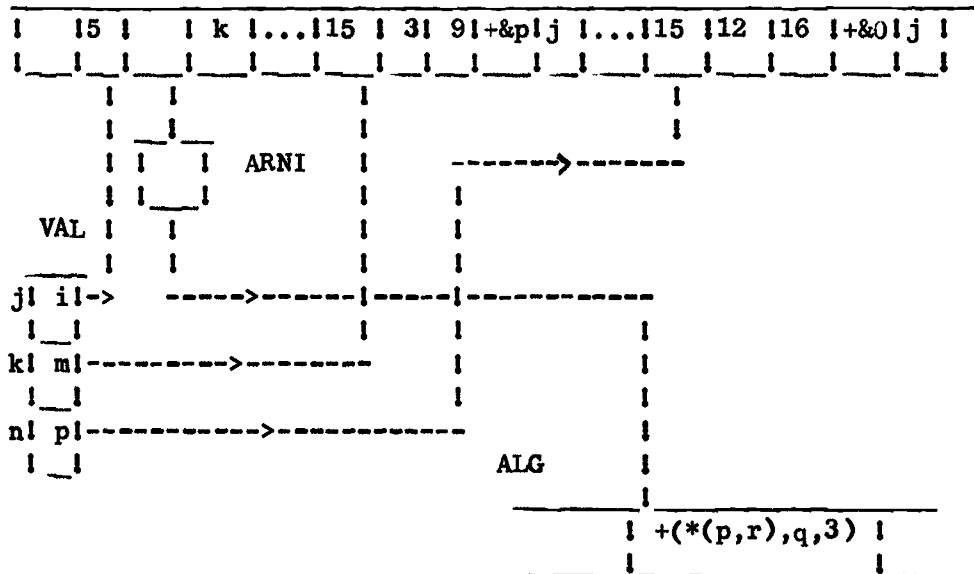


Fig. 3

This example has not shown the use of the operator in INFO(m+3). However, suppose that instead of

a <- p*r,

the instruction was

a <- p+r.

Then x should be changed to

+(p,r,q,3)

and not

+(+(p,r),q,3).

The operator in INFO(m+3) is used to decide whether the leading operator in the new expression may be ommitted. I.e., since INFO(m+3) contains +, +(p,r) is changed to p,r before being substituted.

13.1 Patterns in Polish Form

The process used for patterns conditions is as follows.

1) Call Evalexpr to find the expression, F that is being examined.

2) Call the routine Topatt to put the pattern P into a Polish from.

3) Set j=Contains(F,P) or j = Matchq(F,P), depending on whether the condtion is %contains or %matches.

4) Process the next simple condition (as described in Chapter 9) if j=0 or end=1.

5) Fill in extractors, and then go to next simple condition.

End is a global variable pointing to the arrays EXTR, PTR1, PTR2, and OPR which hold information about extractors. The information is obtained by the recursive functions Contains and Matchq, but cannot be filled in until it is certain that the condition is true. The information they hold is as follows.

EXTR	holds the hash code of the extractor.
PTR1	holds the first position that the extractor points to. (to go in INFO(m+1).)
PTR2	holds the last position that the extractor points to. (to go in INFO(m+2))
OPR	holds the operator that is to go in INFO(m+3).

Topatt is a routine which turns the infix form of the pattern into a Polish form. It is like Evalexpr, but much

simpler, since it does not have any calculation to check for. There are of course more possible basic operands for a pattern than there are for an expression. The following is a list of the operands, together with the code number that is used in the Polish form of a pattern.

%opd	7
%numeric	8
%integer	9
%rational	10
%real	11
%algebraic	12
%any	13

In addition, the operands of an expression, i.e. numeric and algebraic constants may be used. A NAME may have a list of patterns after it, which would be matched against the list of expressions occurring after the name in an expression.

If there is no extractor preceding an operand, the value 7 is put in front of the code. Otherwise, the value 5, and the hash code of the extractor precede the code number. Each pattern is preceded by 7, unless it has an extractor applying to the whole pattern. The following are some examples of patterns and their Polish form.

1) %integer + %real

7	+	0	4	7	9	7	11

2) a _%integer + 5

7	+	0	9	5	81	9	7	1	0	0	0	5

3) a_(%any + %integer)*sin(%real)

+	*	0	17	5	81	+	0	4	7	13	7	9	7	6	40	0	3	5	82	11

where 81 is the hash code of a

82 is the hash code of c

40 is the hash code of sin.

The alternatives of the operands of a (PATTEXPR) are

1) %any
2) %opd
3) %numeric
4) %integer
5) %rational
6) %real
7) %algebraic
8) (PATTEXPR)
9) CONST
10) NAME(PT LIST)?

The details of Topatt are too like Evalexpr to be worth giving.

13.2 Matchq(F,P,op)

Having put both the pattern and the expression into Polish notation, we are in a position to test the condition. The function Matchq, which tests the condition

EXPR '%matches' PATTERN ?

will be described first. It is an integer function whose results are given as follows.

0 if the condition is false.

1 if the condition is true, and the last operand of PATTERN was not %any.

2 if the last operand of PATTERN was %any, but had no extractor preceding it.

2+ if the last operand, %any, had an extractor. The result is two greater than the position in the array EXTR that describes this extractor.

The results are given in this way to allow %any to be applied to two or more consecutive operands. For example, consider the pattern

%algebraic + %any + %integer

matched against

a + b + c + 3

Matchq is used recursively, and in this example it would be called for each of the operands. Thus

Matchq(a,%algebraic) (i)

is called first, and this gives an answer of 1. Then

Matchq(b,%any) (ii)

is called. This is also sucessful. However

Matchq(c,%integer) (iii)

is not. If (ii) gave the answer 1, there would be no way of telling that %any could in fact be applied to b+c as well as to b itself.

If %any were preceded by an extractor, e, it is necessary to know where in the arrays EXTR, PTR1 and PTR2 the representation of e is found. Only then can the correct

value of e (b+c) be entered, as it is found after the call (ii) has been made.

The steps Matchq takes in processing the pattern

P = %algebraic +e:%any + %integer

and the expression

F = a + b + c + 3

will be described first, and then the general description of Matchq will be given. Since the operator of both F and P is +, Matchq is called for each operand.

1) Let Q = 1st term of P (%algebraic)

   G = 1st term of f (a)

2) Matchq(G,Q) = 1.

3) Let Q = next term of P (e:%any)

   G = next term of F (b)

4) Matchq(G,Q) = 2 + end, where end is the position of e in EXTR. Save this in I.

5) Set Q = %integer, G = c.

6) Matchq(G,Q) = 0. However, since I#0, G can be incorporated in %any. Set PTR2(I-2) to point to the end of c. Put the next term of F, (3), into G.

7) Matchq(G,Q) = 1, so the result of the original call is one.

Matchq takes three parameters, the expression F, the pattern, P, and an operator op, which is zero when Matchq is called from Condition. For the general case, let us first consider instances of F and P that have no operators.

1) P = %any, or P = e_%any

If P has an extractor, its name it copied into EXTR(end). PTR1(end) points to the beginning of F, but PTR2(end) is left empty. End is increased by 1, and the result is end+1. (So giving a a position two greater than EXTR.) If there is no extractor, the result is 2.

2) F = NAME or NAME(LIST).

a) P = %opd or %algebraic, e_%opd or e_%algebraic.

If there is an extractor, its hash code is put in EXTR(end). PTR1(end) points to the beginning of F, and PTR2(end) points to the end of F. The result is 1 whether or not there is an extractor.

b) P = NAME or NAME(PATTLIST), or e_NAME or e_NAME(PATTLIST).

The result is 0 unless the names are the same. The result is 1 if both P and F have no list, and 0 unless they both have a list. Then if

F = NAME(F1,F2,...Fn), and

P = NAME(P1,P2,...Pm),

the result is 0 unless m=n and

Matchq(Fi,Pi) = 1 for i=1,2,..n.

If these conditions are satisfied, the extractor, if any, is dealt with as described

in (a), and the result is 1.

c) Result is 0 for any other P.

3) F = CONST.

a) P = %opd or %numeric, e_%opd or e_%numeric.

As for (2a).

b) P = %integer, %rational or %real, e_%integer, e_%rational, or e_%real.

The code number of F is found. If it is suitable for P, proceed as for (2a). Otherwise result = 0. The following defines 'suitable for P'.

code(F) = 1 (integer) is suitable for all three types.

code(F) = 2 (real) is suitable only for %real.

code(F) = 3 (rational) is suitable only for %rational.

c) Result is 0 for any other P.

This completes the discussion for P and F without operators. (Note that P = %any gives the same result if F has operators.)

If P and F have the same leading operator, the terms can be matched against each other, taking note of the case where a term of P is %any, as discussed in the earlier example.

The steps taken in this case are

1) Let I = 0.

2) Set G = 1st term of F, and Q = 1st term of P.

3) J = Matchq(G,F)

4) Go to (7) if J#1.

5) If I>1, the previous term was %any. Therefore this must be 'sealed off'. Set PTR2(I-2) to the end of the previous G, if I>2. Set I = 0.

6) Get the next terms G and Q of F and P, and go to (3). Result is 1 if there are no more terms in F or P. Result is 0 if one is ended, but not the other.

7) Go to (10) if J>0 (i.e. Q was %any).

8) Result = 0 if I = 0, since this means that the previous term was not %any.

9) Result is 0 if this is the last term of F. If not, get the next term, G, and go to (3).

10) 'Seal off' I if it is not zero. Set I = J. Go to (6).

If P and F have different main operators, the result is 0 if op = 0. However if the main operator of P is equal to op, and F does not have an operator, F will be matched against the first term of P. This is to cope with situations like

P = %integer + a_(%algebraic + %real)

F = 1 + b + 0.7.

where the extractor creates a sub-expression in P which is matched against two or more terms in the expression F. This can be seen by considering their Polish forms.

F = +(1,b,0.7)

but

P = +(%integer,a_+(algebraic,%real))

In this case, steps are taken to match b against %algebraic, after Matchq(b,a_+(%algebraic,%real))) has failed.

13.3 Contains(F,P)

The second function, Contains, tests the condition

EXPR ´%contains´ PATTERN ?

It has two parameters, F containing the expression, and P containing the pattern. The results it gives are the same as those for Matchq. A broad outline of the general strategy will be given.

1) If the leading operators of F and P are different, each term of F must be examined to see if it contains P. This is done by calling Contains recursively.

2) If the leading operators are the same, the first term, Q of P is obtained. Then the terms of F are examined until one is found that matches Q. Let us call it G. Obviously the result is 0 if no such term exists. In order for this term to be the first of the required subexpression, if there are n terms in P, there must be at least n-1 terms in F after G, and these must be such that

   Match(Pi,Gi)=1, i=2,3,...n

   where P2,... Pn

   are the second to n-th terms of P,

and G1,... Gn

are the n-1 terms of F that come immediately after G. If this is true, the result of Contains is 1. If not, there may be another term of F beyond G which satisfies the conditions. Hence, beginning at G1, the search for a term matching Q is resumed.

3) (1) and (2) discussed cases where P and F have + and * as their leading operator The case where P=E.P1P2 or P=/P1P2 is simple. If any subexpression of F is of the form E.F1F2 (/F1F2) with Matchq(F1,P1)=1 and Matchq(F2,P2)=1, then the result is 1. Otherwise the result is 0.

4) If P has no operand, each basic operand of F must be matched against P. This is done in the following manner.

Suppose the most general case of F, which is a series of terms added together.I.e.

F = +(F1,F2,...Fn)

Then for each Fi, again suppose the most general case

Fi = *(G1,G2,...Gm)

Each Gj, (j=1,2,...m) can have one of three forms

i) Gj=E.K1K2 (/K1K2). In this case P is matched against both K´s.

ii) Gj has no operator. Then P is matched against Gj.

iii) Gj = +(K1,...Kp), i.e. Gj is a subexpression.

In this case Contains(Gj,P) is called recursively.

The details of associating an extractor with part of F, and of dealing with the case P=%any are very similar to Matchq, and so are not given here.

13.4 Selectq(R,Z)

This routine was mentioned in Chapter 10. Its description has been delayed until the pattern matching mechanism was described. R contains the operand of an expression; in fact it contains the 4th alternative of the operand, which is

'('(SELECTOR)?(EX LB)')'

where

(SELECTOR) = '%rhs'!'%lhs'!
(DESCRIPTOR)'%of'(SELECTOR)?

(DESCRIPTOR) = (EXPR)'%th'(PLB)!(PLB)

(PLB) = (BASICPATT)!(LB)

(EX LB) = (LEXPR)':'(LB)!(FORMULA)

Examples of (SELECTOR)s are

%rhs

4%th %integer %of

%algebraic %of 5%th (%any*%algebraic)

sin(%any) %of

5%th 1:1 %of (where 1:1 contains a pattern.)

If '(EXPR) %th' is ommitted, the first instance is taken. (EXLB) may be an expression, an equation, or a label

expression which gives a statement containing an expression or equation.

Extractors are not allowed in patterns that are used with selectors. If the are found, they will not be connected. The algorithm for Selectq is

1) Set p=length of (SELECTOR).
2) Get the alternative of (EX LB). Go to (9) unless it is a label expression.
3) Get the position g of the label expression in the Storage Tree.
4) If p=1, the selector is %lhs or %rhs, so an equation is required. Otherwise an expression is necessary.
5) Analyse the statement at g as a equation or expression as required, putting the result in Q.
6) Go to (8) unless p=1.
7) If the selector is %rhs, get the expression on the right side of the '=' sign and apply Evalexpr to it, putting the result in Z. Otherwise do the same for the left hand side. Return.
8) Evalexpr Q into Z. Go to (12).
9) If p#1, go to (11).
10) Check that R is an equation. Call Evalexpr to put the appropriate side into Z. Return.
11) Evalexpr R into Z.
12) Return if p=0 (no selector).
13) If the selector is of the form

(EXPR) %th (PLB),

call Eval-to-int for (EXPR), putting the result in J. Otherwise set J=1.

14) Call Topatt for the pattern, putting the result in Q. Set Q=extr_Q, where extr is a dummy extractor used to find the part of the expression that is matched.

15) Save end in endtmp.

16) Do (17) to (19) J times.

17) Fault the program if Contains(Z,Q)=0.

18) PTR2(endtmp) will point to the end of the part of Z that has been matched. If this is not the J-th time, remove the part of Z up to and including the position PTR2(endtmp) gives. If it is the J-th time, remove all but the part contained between PTR1(endtmp) and PTR2(endtmp).

19) Reset end to be endtmp.

20) Return if selector is of the form

(DESCRIPTOR) ′%of′.

(with no selector after it.) Go to (13).

Let us consider an example. Let J 3, P = %integer, and Z=+(a,3,b,4,5,c).

When the cycle is executed the first time, Contains set PTR2(endtmp) pointing to 3. Hence Z becomes

4,b,5,c.

For the second execution of the cycle, PTR2(endtmp) points to 4, so Z becomes

b,5,c.

Finally, for the 3rd time (i.e. the J-th time), PTR1(endtmp) and PTR2(endtmp) point to 5, which is therefore put into Z, to be returned as the result.

XIV Conclusion

AML was an attempt to write an Algebraic Language which incorporated the design features described in Chapter 2. Its outward appearance was planned in 1967, and has not been radically altered. The internal form has gone through many changes, mainly due to the inexperience of the author when she began this project.

Between 1966 and 1969 relatively little was published in the field of Algebraic Manipulation; or rather what was published did little more than amplify previous documents. The last year or so, however, has seen the introduction of several new languages.

These recent languages, taken together, satisfy the criteria laid down in Chapter 2. Some of the ideas they incorporate are more advanced than those that were designed for AML. The semantic pattern matching facilities of Macsyma are an example of this. However, they are in the main the result of many man-years' work, and are based on the experience of the earlier work described in the first part of Chapter 3.

The problems which users of algebraic languages wish to solve tend to be large. Most systems have run into limitations of time or space with some of the problems they have tried to solve. There appear to be two ways of approaching the problem of writing Algebraic Languages. On the one hand, someone with a background in some other field

finds a problem which seems likely to be soluble on a computer. From this a language is written which will solve problems of a class that contains the original problem. Polynomial Manipulators give an example of this type of approach. However it seems inevitable that a problem is found which the original language cannot cope with. This results in a series of extensions which give an untidy, and possibly inefficient language.

The advantages of Polynomial Manipulators are sufficient to make these extremely useful for problems that require large algebraic expressions. In the first place, the rigid format means that only coefficients and exponents of variables need be stored. This method relies on the fact that polynomials have a Canonical form, so that the problem of equality reduces to a test for an identical match. W.S.Brown, [30] pages 195-211 and G.E.Collins, [30] pages 212-222 have studied methods for finding the greatest common divisor of two polynomials. Caviness [31] has proved that Rational Functions and also the class of radical expressions with the exponential function and restricted composition (i.e. exp(x) is in the class, but exp(exp(x)) is not.) have canonical forms.

CAMAL, ALTRAN and MATHLAB all manipulate Rational Functions, where a Rational Function is expressed as the quotient of two polynomials.

The second approach is to start off with a blueprint for a language that will suit everybody. Unfortunately this

may well mean that no problems of significant size can be solved.

Richardson [32] has shown that for the class of expressions generated by

a) The rational numbers, PI and log2,

b) The variable x,

c) The operations of addition, multiplication and composition,

and

d) The functions sin, exp, and the absolute value function,

the predicate E = 0 is recursively undecidable. This means that no Canonical form can be found for expressions in this class. Hence the simplification algorithms of the various languages differ, since the interpretation the word 'simplify' has a different meaning to different people. The representation of an expression has no well defined form either.

Most of the sucessful languages for general expressions are written in LISP:- REDUCE, SCRATCHPAD, and MACSYMA. Because the same implementing language is used, the writers of the more recent languages have been able to borrow heavily from the older work, and what is emerging is a system of languages from which the user can pick out the facilities most useful to him.

Every language has this problem of generalisation. It seems impossible to have available all the tools required by

all classes of user. Hence the tendency has been to endeavour to supply the user with means by which he can extend the language himself. There does not seem to be a language that has solved the problem completely; most give a facility for incorporating new routines written in the host language. However this does not seem satisfactory for the ordinary user.

Formula Algol's answer was to provide pattern matching and let the user do the rest. This is carrying things to an extreme, as the average user does not want to waste time defining the many different forms of the Distributive Law, to take one example. However I think that the initial idea is the right one. One approach would be the following:

1) Have a highly efficient pattern matching facility which is also understandable by the general user.
2) Using this, write a library of the basic algebraic functions; the user can add to these if necessary.

If a large library is to be built up, then it is obvious that an interpreter should not be used on the routines stored there. Ideally they should be in machine code. Since the idea is to write them in an algebraic language, this means that a compiler must be written for that language. I visualise that both an interpreter and a compiler would be available, thus giving the user the option of compiling the parts of his program that he did not wish to alter. This could be done by a statement of the form

%compile 1:1 %into C1.

This system requires that the structure of AML is revised considerably. For efficiency, library routines should be compiled rather than interpreted. However to write a compiler for AML as it stands would not be possible, because the labelling system allows one to write self-modifying programs.

The structure of AML is also unsatisfactory for several other reasons. The self - modifying aspect of the language, i.e. the fact that a labelled statement can be used as both data and program is like the facility offered in assembly coding, and requires the user to organise his statements very carefully. This can be confusing, particularly in large programs.

The labelling system provides the user with a virtual memory that can be finely divided. Apart from this feature the virtual memory is like the address space of a computer devoid of indirection, indexing and re-location. The user who wishes to use recursion or mutual recursion has to create his own stack and pointer mechanism. Routines that are used together must use mutually exclusive sets of addresses. Code is not relocatable in this virtual memory. The first property creates an onerous task for the user, the second and third present problems for the construction of libary facilities.

The flow of control through an AML program is likely to be very complicated. This is another fault of assembly languages that high level languages are intended to

overcome. The fact that an algebraic formula can be regarded as a labelled statement or as the value of an variable is also undesirable. It gives an assymetry that is unpleasing, confusing, and which creates problems in the implementation

For all these reasons, the basic structure of AML should be re-examined and reorganised.

All labels of blocks of compiled code would begin with a C, and th block would be accessed by %do, e.g.

%do C1,

or by a routine declaration.

%routine alpha %at C1.

Conventions would have to be established for handling parameters.

Library routines would automatically be compiled, and the user would be able to declare a routine to be a library routine. From then on he would have that routine in his own private library.

The idea put forward at the beginning of this thesis, i.e. that simplification should not take place unless requested should, I think, be adhered to. However it becomes very annoying to have to write

a = (b + c)*c

%distrib a, %simplify a

for each assignment, when there are a number of assignments that the user wants in simplified form. Therefore, instead of (or as well as) the commands there should be flags that can be set. Initially all would be off. However if the user were to put on, for example, the 'simplify' flag, then any assignment would cause the expression on the right to be simplified before being stored.

Moving from the external appearance to the internal form, it is obvious that arrays are not flexible enough to be used for algebraic expressions. Therefore an improvement

would be to have a free list, and use elements of this to build up the algebraic formulae. The basic structure, i.e. the Polish notation described earlier would be kept. The list would consist of two cells; a byte to hold the actual character, and a short integer to point to the next cell required. This of course would be three times as long as the present structure. However it may be possible to have pointers to common subexpressions and to economise by reducing the number of copying operations required.

Another saving, in this case of time, could be made by using canonical forms of expressions where possible. Since the internal form uses hash coding, this would be quite easy to set up. The advantage of canonical forms is that every comparison required is merely a test for exact equality. However it is often confusing to the user if his expressions are given back in a different order; particularly if the new order is as arbitary as that imposed by hash coding. Two methods could be used to overcome this: the initial order could be remembered, and the expression output in something close to this. Also the user could be allowed to specify the order he requires, as done in REDUCE. There are, however, severe problems with syntactic pattern matching if Canonical Forms are used. Again MACSYMA's semantic patterns seem to provide the answer.

Given this reorganisation, it seems that AML will satisfy most of the demands that are made on an Algebraic Language. The examples in Appendix E show how most of the

facilities that appear to be missing can be written in quite short routines.

Other facilities that should be incorporated in AML and are not easily obtained with the language as it is, are the capablity of defining new operators, and the provision for classes of objects that satisfy a particular property.

This thesis describes the design and implementation of an algebraic language. It can be seen from the conclusion that the result, although workable, is not completely satisfactory. However the problem was an exacting one, and only be exploring it in this depth could solutions be found. Many wrong turnings were taken before the language as it stands was developed, and it is only since it has been available on EMAS (from February of this year) that its true nature could be seen. AML has not given a complete answer to the problem, but hopefully it has made a significant step in that direction.

Appendix A - The Syntax of AML

The Built in Phrases

```
                <name>
                <const>
                <expr>
                <pattexpr>
                <lb>
                <primes>
                <list>
                <text>
```

The Syntax

```
<activest>      = ´%continue´!
                  ´%local´<name>*!
                  ´%empty´<name>*!
                  ´%local´?´%array´<ardefn>*!
                  <uncondst>´%if´<condition>!
                  <uncondst>´%unless´<condition>!
                  <iu><condition>´%then´<elsecl>!
                  ´%local´?<rt><name><fpp>?´%at´<label>!
                  <uncondst>
<uncondst>      = ´%return´!
                  ´%finish´!
                  ´%exit´!
                  ´%stop´!
                  ´%result=´<expr>!
                  ´%print´<text>!
```

```
                <name>'<-'<expr>!

                '%long'?'%real'?<name><list>?'='<expr>!

                <name><list>?!

                <command>*<untilcl>?

<formula>     = <expr>'='<expr>!<expr>

<iu>          = '%if'!'%unless'

<elsecl>      = <uncondst>'%else'<uncondst>!<uncondst>

<untilcl>     = '%until'<condition>!

                '%while'<condition>!

                '%for'<name>'='<expr>','<expr>','<expr>

<opd>         = <lb>!<const>!<name><plist>?

<exlb>        = <lb>!<formula>

<plist>       = <primes><list>

<selector>    = '%rhs'!'%lhs'!

                <descript>'%of'<selector>?

<descript>    = <expr>'%th'<plb>!<plb>

<plb>         = <lb>!<basic patt>

<ardefn>      = <name>*'('<boundpr>*')'

<boundpr>     = <expr>':'<expr>

<rt>          = '%routine'!'%function'

<fpp>         = '('<fp>*')'

<fp>          = <fpdelim><name>*

<fpdelim>     = '%value'!'%name'!'%array'!

                '%fn'!'%routine'!'%label'

<condition>   = <sc>'%and'<andsc>!<sc>'%or'<orsc>!<sc>

<andsc>       = <sc>'%and'<andsc>!<sc>

<orsc>        = <sc>'%or'<orsc>!<sc>
```

```
<sc>            = '('<condition>')'!<expr>'//'<expr>!
                  <name><list>?'%matches'<patlb>!
                  <name><list>?'%contains'<patlb>!
                  <expr><comp><expr><rest cexpr>?!<label>
<rest cexpr>    = <comp><expr>
<comp>          = '<='!'>='!'<'!'>'!'#'!'='
<command>       = '%do'<label>*?!
                  '%write'<label>*?!
                  '%erase'<label>*?!
                  '%label'<label>*?'%as'<label>*!
                  '%copy'<namelist>*'%into'<label>*!
                  '%read'<namelist>*'%from %file'<expr>!
                  '%print' '%no'? '%results'<names>*?!
                  '%eval'<namelist>*!
                  '%long'?'%real'?'%simplify'<intocl>?!
                  '%distrib'<intocl>?!
                  '%expand'<intocl>?<term>?!
                  '%subs'<forcl>'%in'<intocl>?!
                  '%diff'<n>?<intocl>?!
                  '%addsum'<intocl>?
<term>          = '%for'<expr>'%th %term'!'%to'<expr>'%terms'
<forcl>         = <expr>*'%for'<expr>*!<namelist>*<equation>*
<namelist>      = <name><list>?
<equation>      = <expr>'='<expr>!<lb>
<patt>          = <name>'_'<basicpatt>!<basicpatt>
<basicpatt>     = '%integer'!'%real'!'%rational'!
                  '%algebraic'!'%opd'!'%numeric'!'%any'!
```

```
                    ´%factor´!´%term´!´(´<pattexpr>´)´!
                    <const>!<pname><ptlist>?
<pname>           = ´%name´!<name>´_ %name´!<name>
<intocl>          = <names>*<wrt>?´%into´<label>*!
                    <names>*<wrt>?
<names>           = <lb>!<formula>
<patlb>           = <lb>!<pattexpr>
<wrt>             = ´%wrt´<expr>
<lexpr>           = ´(´<expr>´)´!<const>!<name><list>?
<op>              = ´**´!´/´!´*´!´-´!´+´!´&´!´!´!´!!´!´<<´!´>>´
<n>               = ´(´<expr>´)´
<ptlist>          = <primes>´(´<pattexpr>*´)´
<label>           = <lb>!<const>
```

Appendix B - Average Length of Lines in Sample AML Programs

1) No. of characters = 523

   No. of statements = 32

   Average no. of characters/statement = 16

2) No. of characters = 351

   No. of statements = 21

   Average no. of characters/statement = 16

3) No. of characters = 487

   No. of statements = 30

   Average no. of characters/statement = 16

4) No. of characters = 606

   No. of statements = 33

   Average no. of characters/statement = 18

5) No. of characters = 926

   No. of statements = 46

   Average no. of characters/statement = 20

6) No. of characters = 399

   No. of statements = 26

   Average no. of characters/statement = 15

7) No. of characters = 392

No. of statements = 20

Average no. of characters/statement = 19

8) No. of characters = 1194

No. of statements = 50

Average no. of characters/statement = 23

9) No. of characters = 118

No. of statements = 11

Average no. of characters/statement = 10

10) No. of characters = 111

No. of statements = 8

Average no. of characters/statement = 13

11) No. of characters = 79

No. of statements = 8

Average no. of characters/statement = 9

12) No. of characters = 127

No. of statements = 9

Average no. of characters/statement = 14

13) No. of characters = 547

No. of statements = 29

Average no. of characters/statement = 19

14) No. of characters = 199

No. of statements = 10

Average no. of characters/statement = 19

15) No. of characters = 265

No. of statements = 23

Average no. of characters/statement = 11

16) No. of characters = 683

No. of statements = 30

Average no. of characters/statement = 22

17) No. of characters = 300

No. of statements = 18

Average no. of characters/statement = 16

18) No. of characters = 158

No. of statements = 11

Average no. of characters/statement = 14

19) No. of characters = 338

No. of statements = 18

Average no. of characters/statement = 18

20) No. of characters = 116

No. of statements = 5

Average no. of characters/statement = 23

APPENDIX C - COMPARISON OF IMP & AML SYNTAX ANALYSERS

STATEMENT

%LOCAL %ARRAY A(1:10)

TIME FOR IMP TO ANALYSE IT 0.086520000000

TIME FOR AML TO ANALYSE IT 0.039990000000

STATEMENT

%DO 1

TIME FOR IMP TO ANALYSE IT 0.139990000000

TIME FOR AML TO ANALYSE IT 0.079990000000

STATEMENT

%DO 1:1:1 %FOR I=1,1,10

TIME FOR IMP TO ANALYSE IT 0.459970000000

TIME FOR AML TO ANALYSE IT 0.279980000000

STATEMENT

RTN(A,B,C,D)

TIME FOR IMP TO ANALYSE IT 1.919870000000

TIME FOR AML TO ANALYSE IT 0.239980000000

```
STATEMENT

%IF I=1 %THEN %DO 1 %ELSE P=Q
TIME FOR IMP TO ANALYSE IT  0.639950000000

TIME FOR AML TO ANALYSE IT  0.179990000000
```

```
STATEMENT

 I=P+Q*R(1,2)+(X+Z)*O
TIME FOR IMP TO ANALYSE IT  0.739950000000

TIME FOR AML TO ANALYSE IT  0.279980000000
```

```
STATEMENT

%ARRAY A,B,C(1:10,2:20),D,E(1:20)
TIME FOR IMP TO ANALYSE IT  0.199980000000

TIME FOR AML TO ANALYSE IT  0.159990000000
```

Appendix D

The following are the built in functions and routines of AML.

1) Routine space, spaces(n)

   Output 1 space, n spaces.

2) Routine newline, newlines(n)

   Output 1 newline, n newlines.

3) Routine write(a), write(a,m), write(a,m,n)

   If a is an algebraic expression, output it. N and m are ignored.

   If a is an integer, output it, taking up m+1 positions (the extra place is for the sign). If m is not given, take m to be 6. N is ignored.

   If a is a rational, output it, taking m+1 positions for the numerator, and n+1 positions for the denominator. Default is m=6, n=6.

   If a is a real or long real, output it in fixed point form, taking m+1 positions before the decimal point, and n positions after it. Default is m=6, n=6.

3) Routine getfile(f), getfile(f,i)

   Read in the file whose position in the input stream is given by the integer f. This file is expected to contain labelled and unlabelled statements of AML. Store the labelled statements, and execute the

unlabelled ones, exactly as if coming from the teletype. Print the statements at the console, unless i=0.

4) Function sin(x)

If x is an algebraic expression, this gives the symbolic function sin(x). Otherwise it evaluates sin(x) to a real number.

5) Functions cos(x), tan(x), cotan(x), sec(x), cosec(x), exp(x), log(x)

As for (5).

6) Functions intpt(x),fracpt(x)

As for (4) if x is an algebraic expression. For x numeric,

Intpt(x) gives the integer part of x, [x].

Fracpt(x) gives the real number x-[x].

7) Functions power(a,x,r,i,j), sigma(a,r,i,j)

Symbolic notations for summation series.

a is an algebraic expression (possibily dependent on r)

x is an algebraic constant

r is an algebraic constant

i is an integer

j is an integer

power represents

$$\sum_{r=i}^{j} a*x**r$$

sigma represents

$$\sum_{r=i}^{j} a$$

8) inf - a reserved name standing for infinity (∞)

9) pi - a reserved word standing for π (=3.141593)

10) exp - a reserved name standing for e(=2.718281). See also (5).

11) %routine collect(%value a,b,%name c,d)

This routine splits the expression a into the form

a -> b*c + d   (i)

It has been found that a routine which returns the two parts c and d is more useful than a command which merely rearranges the expression into the form given by (i). Obviously the coefficient of b in the expression a is given by c. The value of b may be any expression.

Examples

```
(i)   a = p*x*q + y*q*p + z + x*r*s + y*s*r
        + p*q*(f + g) + r*f*s + s*r*g
      b = p*q + r*s
Gives c = x + y + f + g
      d = z
(ii)  a = (p + q)*x*y + z*p + r*s*(z+y) + q*z
      b = p + q + r*s
Gives
      c = x*y + z
```

```
       c = x*y + z

       d = 0

iii)   a = x + y + z

       b = p + q

Gives

       c = 0

       d = x + y + z

(iv)   a = p + q + r

       b = p + q

Gives

       c = 1

       d = r

(v)    a = power(r**2,x,r,0,10) + p*x**3

       b = x**3

Gives

       c = p+9

       d = power(r**2,x,r,0,2) +

                 power(r**2,x,r,4,10)
```

Appendix E - Examples

1) The routine CHECK3 replaces sub expressions of the form

sin(3a)

and cos(3a)

by the equivalent expressions

3sin(a)-4sin(a)**3

-3cos(a)+4cos(a)**3

in the expression passed to it as a parameter.

```
!CSC013          J.OFFICER RCC SHELVES        01/12/71   08.30.26

           AML  UPDATED  25/11/71

!         EXAMPLE 1

          %routine check3(%name p) %at 1:1

1:1       %local a,b

:1:1       %do 1:2 %if p %contains a_sin(3*b_%any)

1:2        a<-3*sin(b)-4*sin(b)**3

1:1:2      %do 1:3 %if p %contains a_cos(3*b_%any)

1:3        a<- -3*cos(b)+4*cos(b)**3

1:1:3      %simplify p

:4         %return

           q=sin(3*x)+sin(x)+cos(3*x)-3*cos(x)**3

           check3(q)

4*sin(x) - 4*sin(x)**3 - 3*cos(x) + cos(x)**3

           %finish
```

2) A function for evaluating determinants.

```
!CSC013          J.OFFICER RCC SHELVES        01/12/71    09.17.18

          AML  UPDated  25/11/71

!     example 2

     %print %no %results

     %array r(1:4,1:4)

     %function det(%array r, %value i) %at 1:1

1:1  %do 2:1 %if i=2

1:1:1  %local  %array s(1:i - 1,1:i - 1)

1:1:2  %local p,j,k,l,m,n

1:1:3 p=0

1:1:4  %do 2:2 %for j=1,1,i

1:1:5  %distrib p, %simplify

1:1:6  %result =p

1:7  %do 1:8 %for i=1,1,k

1:7:1 q=det(r,k)

1:7:2  %print ´q=´;write(q);newline

1:8  %do 1:9 %for j=1,1,k

1:9  %read r(i,j) %from  %file 1

2:1 p=r(1,1)*r(2,2) - r(1,2)*r(2,1)

2:1:5  %simplify p

2:1:6  %result =p

2:2 n=0

2:2:1  %do 2:3 %for k=2,1,i
```

```
2:2:2 p=p + ( - 1)**(j - 1)*r(j,1)*det(s,i - 1)

2:3 m=1;n=n + 1

2:3:1  %do 2:4 %for l=1,1,i

2:4  %do 2:5 %unless l=j

2:5 s(m,n)=r(l,k)

2:5:1 m=m + 1

k = 3; %do 1:7

da db dc

dd de df

dg dh di

q=da*de*di - da*df*dh - dd*db*di + dd*dc*dh + dg*db*df - dg*dc*de

%finish
```

3) The first seven Legendre Polynomials, using the relation

Pn(x) = (1/n)*(2n-1)xPn-1 - (n-1)Pn-2(x)

P0(x) = 1, P1(x) = x.

This example is given in [26] and [12].

```
!CSC013        J.OFFICER RCC SHELVES        02/12/71   21.01.21


        AML  UPDATED  25/11/71

      example 3

%print %no %results

%array p(0:10)

p(0)=1;p(1)=x

1:1 p(n)=1/n*((2*n-1)*x*p(n-1)-(n-1)*p(n-2))

:1:1 %distrib p(n),%simplify

:2 write(p(n))

%do 1:1 %for n=2,1,7

 - 1/2 + 3/2*x**2

 - 1/3 - 1/6*x + 5/2*x**3

 - 5/8 - 7/12*x + 5/6*x**2 + 35/8*x**4

 - 7/15 - 151/120*x - 21/20*x**2 + 7/2*x**3 + 63/8*x**5

 - 11/16 - 161/120*x - 129/80*x**2 - 77/40*x**3 + 161/16*x**4
 + 231/16*x**6

 - 19/35 - 1319/560*x - 407/120*x**2 + 3/560*x**3 - 143/40*x**4
 + 407/16*x**5 + 429/16*x**7

%finish
```

4) The function SIGEVAL turns all power series functions of the form

´power(A(r),x,r,m,n)´

into the sum of terms that the function represents. If m or n are infinite (-INF or INF), finite approximations are taken from the integers i and j.

```
!CSC013        J.OFFICER RCC SHELVES        01/12/71   13.07.01

          AML  UPDated  25/11/71

!      example 4
       %print %no %results
10     %function h(%value n,r) %at 1:1
1:1  %local k,res
1:1:1  %result =n %if r=0
1:1:2 res=n
1:1:3  %do 1:1:5 %for k=1,1,r - 1
1:1:4  %result =res
1:1:5 res=res*(n - k)
     %function sigeval(%value s,i,j) %at 2:1
2:1  %local a,b,c,d,e,p,q,t,k
2:1:1  %result =s %unless s %contains p_power(a_ %any ,b_ %algebraic ,:
c_ %algebraic ,d_ %numeric ,e_ %numeric )
2:1:2 q=0
2:1:2:2 d=i %if d= - inf
2:1:2:3 e=j %if e=inf
2:1:2:4  %result=s %unless d %matches %integer %and e %matches %integer
```

```
2:1:3  %do 2:1:6 %for k=d,1,e

2:1:3:5  %do 10

2:1:3:6  %eval q

2:1:4  %simplify q;p< - q

2:1:5  %result =s

2:1:6 t=a

2:1:6:1  %subs k %for c %in t

2:1:6:2 q=q + t*b**k

3:1 s=power(h(n,r)*a**r,x,r,0,p)

4:1  %do 3:1

4:2 t=sigeval(s,0,0);write(t);newline

     n=1/2;a=2;p=5

     %do 4

1/2 - 1/2*x - x*2 + 3*x*3 - 15*x*4 + 105*x*5

%finish
```

5) The f and g series of Celestial Mechanics. The calculation of these values has become almost a standard test for Algebraic Systems. Fi and Gi are sequences of polynomials in the time dependent variables mu, s and e. They satisfy the relations

$$Fi = F'i-1 = muGi-1$$

$$Gi = Fi-1 - G'i-1.$$

F0 and G0 are 1, and mu, s and e are connected by the equations

$$mu' = -3*mu*s$$

$$s' = e - 2*s**2$$

$$e' = -s*(mu +2*e)$$

Examples of this calculation are given in [26] and [12]. The original calculation is described in [28]. (The ALGEBRAIC STACK overflows during the calculation of the 5th f and g.)

```
!CSC013          J.OFFICER RCC SHELVES          06/12/71   20.26.19

            AML  UPDated  25/11/71
!     example 5
%array f,g(0:10)
%print %no %results
x1=-s*(mu+2*e)
x2=e-2+s**2
x3=-3*mu*s
f(0)=1;g(0)=1
```

```
1:1 f(i)

3:1 g(i)

5:1 0

 :2 0

 :3 0

6:1 0

 :2 0

 :3 0

10:1 f(i) = -mu*g(i-1)+x1*5:1+x2*5:2+x3*5:3

  :2 g(i) = f(i-1)+x1*6:1+x2*6:2+x3*6:3

  :3 %distrib f(i),g(i),%simplify

  :3:4 %print ´f(´;write(i,1)

    :5 %print ´) =´;write(f(i));newline

    :6 %print ´g(´;write(i,1)

    :7 %print ´) =´;write(g(i));newlines(2)

10:4 %do 14 %for j=1,1,3

12:1 e

  :2 mu

  :3 s

14  %diff 1:1 %wrt 12:j %into 5:j

14:1 %diff 3:1 %wrt 12:j %into 6:j

%do 10 %for i=1,1,4
f( 1)= - mu

g( 1)=   1

f( 2)=2 - mu - e - s**2

g( 2)= - mu

f( 3)=2 + mu**2 + s*mu + 2*s*e - e - s**2 + 6*mu*s**2
```

```
g( 3)=4 - mu - 2*e - 2*s**2

f( 4)= - 7*mu - mu**2 + 4*mu*e - 4*mu*s**2 + 4*s*mu + 2*s**2*e

+ 3*s*e - 2*s - 12*s**2 + s**3 + 6*s**4 - 3*mu**2*s - 6*mu*s*e

- 36*mu**2*s**2

g( 4)=4 + mu**2 + 3*s*mu + 6*s*e - 2*e - 2*s**2 + 18*mu*s**2

%finish
```

6) The routine ´solve´ expects a polynomial of degree two in f. It finds the roots of the quadratic equation f(x)=0, where x is given by the second parameter. The results are returned in the %name type parameters s1 and s2.

```
!CSC013          J.OFFICER RCC SHELVES          06/12/71   20.56.10

          AML  UPDated  25/11/71

!     example 6
%routine solve(%value f,x,%name s1,s2) %at 24
24:1 %local a,b,c,d
:2 collect(f,x,b,d)
:3 collect(d,x**2,a,c)
:4 d=(b**2-4*a*c)**(1/2)
:5 s1=(-b+d)/(2*a)
:6 s2=(-b-d)/(2*a)
:7 %return
g=2*x**2+7*x+3
2:1 solve(g,x,p,q)
 :2 %print ´p =´;write(p);newline
 :3 %print ´q = ´;write(q);newline
%do 2
p=  -1/   2
q=  -3
```

```
g=a*x**2+3*a*x-6

%do 2

p=( - 3*a + ((3*a)**2 - 4*a* - 6)**1/2)/(2*a)

q=( - 3*a - ((3*a)**2 - 4*a* - 6)**1/2)/(2*a)

%print %no %results

%expand p,q,%distrib,%simplify

write(p);newline

3/2 - 1/2*a** - 1 + 1/2*a** - 1*((3*a)**2 - 4*a* - 6)**1/2

write(q);newline

3/2 - a** - 1 + 1/2*a** - 1*((3*a)**2 - 4*a* - 6)**1/2

%finish
```

7) A routine to truncate a polynomial in x to powers of i and less. A contains the polynomial.

```
!CSC013         J.OFFICER RCC SHELVES        07/12/71   21.51.34

          AML  UPDATED  25/11/71

!     example 7
%print %no %results
%routine truncate(%name a,%value i) %at 20
20:1 %local b,p,q,r,t
  :2 %do 20:10 %until b=0
  :1:5 b=a;a=0
20:10:2 q=0
     :3 p=b %unless b %matches p_%any+q_%any
     :4 contains(p,r,20:20)
     :5 %if r#0 %then %do 20:11 %else a=a+p
     :6 b=q
20:20 x**s_%integer
20:11:1 a=a+p %unless s>i
20:4 %simplify a
  :5 %return
%routine contains(%name a, r,%label l) %at 23
23:2 %local p,q,z
  :3 p=1;r=a;q=1
  :4 r=0 %unless r %contains z_l
  :5 %return %if r=0
  :6 z<-1;%simplify r;%return %if a %contains l
y=x**9+x**7+x**5+x**3+x+1
```

```
truncate(y,5)

write(y)

1 + x**5 + x**3 + x

%finish
```

8) A routine to find the coefficient of a given power of x, in a expression that involves the AML function 'power'.

```
!CSC013        J.OFFICER RCC SHELVES        07/12/71   20.57.58

          AML  UPDATED  25/11/71

! example 8
%print %no %results
%function power coeff(%value z,x,c) %at 22
22:2 %local f,s,r,i,j,d,b,p,q,t
:3 contains(z,p,22:20)
:4 %result=0 %if p=0
:5 contains(p,q,22:21)
:6 %result=0 %if q=0
:7 b=c-d;%simplify b
:8 %result=0 %unless b %matches %integer
:9 %result=0 %unless i<b<j
:10 f=t*q;%subs b %for s %in f,%simplify
:11 %result=f
:20 power(t_%any,x,s_%algebraic,i_%numeric,j_%numeric)
:21 x**d_%any
%routine contains(%name a, r,%label 1) %at 23
23:2 %local p,q,z
  :3 p=1;r=a;q=1
  :4 r=0 %unless r %contains z_1
  :5 %return %if r=0
  :6 z<-1;%simplify r;%return %if a %contains 1
a=power(r**2,x,r,1,10)*x**(i-3)
```

```
y=powercoeff(a,x,i+1)
write(y)
  16
%finish
```

9) This example finds the first five terms of the expansion of exp(sin(y)). The same example is given in [25].

```
!CSC013         J.OFFICER RCC SHELVES        08/12/71   08.28.50

          AML  UPDATED  25/11/71

!     example 9

%print %no %results

siny = y; fact = 1; expy = 0

1:2  %if 4//(n - 1) %then i=1 %else i= - 1

1:2:1 fact=fact*(n - 1)*n

1:2:2 siny=siny + i*y**n/fact

1:3 fact=fact*n

1:3:1  %real z=siny**n/fact

1:3:2  %expand z, %simplify

1:3:3 expy=expy + z

1:4 collect(q,y**i,p,r)

1:4:1 expy=expy + p*y**i

1:4:2 q=p

3  %do 1:2 %for n=3,2,7

3:1:1:2  %expand siny, %simplify

3:1:1:7 write(siny);newline

3:1:5 fact=1; %do 1:3 %for n=1,1,5

3:2  %simplify expy

3:3 collect(expy,y,p,q)

3:4  %do 1:4 %for i=2,1,5

3:5 expy=r + expy

%do 3
```

```
y + 1/6*y**3 + 1/120*y**5 - 1/5040*y**7

%print 'result =';write(expy)

result= y + 1/6*y**3 + 1/120*y**5 - 1/5040*y**7 + 0.500000*(y**2
 + 2*y*0.166666*y**3 + 2*y*0.8333*y**5 + 2*y* - 47.953109*y**7
 + 0.27777*y**6 + 0.2777*y**8 + 0.3*y**10 - 3208.316894*y**12
 + 0.0*y**14)

%finish
```

10) A function to multiply two power series P and Q. P must range from 0 to some positive integer, and Q must have the same range. The array C holds the coefficients of the result.

```
!CSC013        J.OFFICER RCC SHELVES        08/12/71   13.35.38

          AML  UPDATED  25/11/71

!      example 10
%print %no %results
%function powermult(%value p,q,%array c) %at 1:1
1:1:5  %result =p*q %unless 1:1:20 %and 1:1:21
1:1:5:5  %result =p*q %unless y=x %and m=n
1:1:6  %do 1:2 %if s=r
1:1:7  %do 1:3 %for i=0,1,n
1:1:8  %result =power(c(r),x,r,0,n)
1:1:20 p %matches power(a_%any,x_%algebraic,r_%algebraic,0,n_%integer)
1:1:21 q %matches power(b_%any,y_%algebraic,s_%algebraic,0,m_%numeric)
1:2  %empty s; %subs s %for r %in b
1:3 t=0
1:3:1  %do 1:4 %for j=0,1,i
1:3:2  %distrib t,%simplify;c(i)=t
1:4 v=a*b
1:4:1  %subs j,i - j %for r,s %in v
1:4:2 t=t + v
%array c(0:20)
p=power(3*r+a,x,r,0,5)
q=power(k*s**2,x,s,0,5)
```

```
t=powermult(p,q,c)

%print 't =';write(t);newline

t=0 + (3*0 + a)*k*6**2 + (3*1 + a)*k*5**2 + (3*2 + a)*k*4**2

2:1 %print 'c(';write(i,1);%print') ='

 :2 write(c(i));newline

    %print 'coefficients:';newline

coefficients:

    %do 2 %for i=0,1,5
c( 0)=   0

c( 1)=k*a

c( 2)=5*k*a + 3*k

c( 3)=14*k*a + 18*k

c( 4)=30*k*a + 60*k

c( 5)=55*k*a + 150*k

    %finish
```

Appendix F - References.

[1] Communications of the Association for Computing Machinery, Volume 9, number 8. (1966)

[2] Bobrow, D. G. (Ed) Symbol Manipulation Languages and Techniques (Proceedings of the IFIP working conference on Symbol Manipulation Languages, Pisa). North-Holland Publishing Company, Amsterdam. (1968)

[3] Van de Riet, R. P. Formula Manipulation in ALGOL 60. (Preliminary report) Report no. TW 101, Stiching Mathematisch Centrum, Amsterdam (1966)

[4] Formula Algol User's Manual, no CCU - 73. Carnegie Mellon University, Pittsburgh. (1967)

[5] Sammett, J. E. Programming Languages: History and Fundementals. Prentice - Hall, Inc. Englewood Cliffs, New Jersey. (1969)

[6] FORMAC. Report no. 24, Edinburgh Regional Computing Centre. (1970)

[7] AFIPS Fall Joint Computer Conference, Volume 27, part I. Spartan Books, Washington (1965)

[8] Brown, W. S, Hyde, J. P, Tague, B. A. The Alpak System. Bell System Technical Journal, Volume 42, no. 5 (1963), Volume 43, no, 2. (1964)

[9] Brown. W. S. Altran revisted. Bell Telephone Laboratories Incorporated, Murray Hill, New Jersey. (1969).

[10] IFIPS International Conference, Edinburgh. (1968)

[11] Martin, W. A. Symbolic Mathematical Laboratory. Thesis, M.I.T. no MAC TR-36. Project MAC, Cambridge, Mass. (1967)

[12] Hearn, A. C. Reduce 2. Decus Programming Library, no 10-21a, Digital Equipment Corporation. (1971)

[13] Bourne, S. R, Horton, J. R. The Camal System Manual, Computer Laboratory, University of Cambridge, England. (1971)

[14] Barton, d. A scheme for Manipulative Algebra on a Computer. Computer Journal, Volume 9, no. 4. (1967)

[15] Barton, D, Bourne, S. R, Burgess, C. J. A Simple Algebra System. Computer Journal, Volume 11, no. 3. (1968)

[16] Barton, D, Bourne S. R, Fitch, J. P. An Algebra System. Computer Journal, Volume 13, no. 1. (1970)

[17] Barton, D, Bourne S. R, Horton J. R. The Structure of the Cambridge Algebra System. Computer Journal, Volume 13, no. 3. (1970)

[18] D'Inverno, R. A. ALAM programmers' Manual Dept. of Mathmatics, King's College, London.

[19] D'Inverno R. A. ALAM - Atlas Lisp Algebraic Manipulator. Computer Journal, Volume 12, no. 2. (1968)

[20] Griesmer, J. H, Jenks, R. D. Scratchpad/1. I.B.M. Thomas J. Watson Research Center, Yorktown Heights, New York. (1970)

[21] Korsvold, K. An Online Algebraic Simplifying Program

Stanford Artificial Intelligence Project. Memo 37, Stanford University, California. (1965)

[22] Moses, J. Symbolic Integration. Thesis, M.I.T, no. MAC-TR-47. Project MAC, Cambridge, Mass. (1967)

[23] Shaw, J. C. JOSS. AFIPS Fall Joint Computer Conference, Volume 26, part I. Spartan Books, Washington. (1964)

[24] Perlis, A. J. LCC - Language for Conversational Computing. no. CRD-8 Carnegie Mellon University, Pittsburgh. (1967)

[25] Lapidus, A, Goldstein, M. Some Experiments in Algebraic Manipulation by Computer. Communications of the A,C,M, Volume 8, No. 8. (1965)

[26] Barton, D. Bourne, S. R, Fitch, J. P, Horton, J. R. Some Applications of the Cambridge Algebra System. Computer Laboratory, University of Cambridge. (1971)

[27] Piaggio, H. T. H. Differential Equations. G. Bell and sons, London. (1943)

[28] Sconzo, P, Leschack, A. R, Tobey, R. Symbolic Computation of the f and g Series by Computer. Astronomical Journal, Volume 70, No. 4. (1965)

[29] Goldstein, M, Thaler, R. M. Bessel Functions For Large Arguments. Mathematical Tables and Other Aids to Computation, XII, No 61. (1958)

[30] Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation (1971)

[31] Caviness, B. F. On Canonical Forms and

Simplifications. Doctoral Dissertation, Carnegie Mellon University, Pittsburgh. (1968)

[32] Richardson, D. Some Unsolvable Problems Involving Functions of a Real Variable. Notices of the American Mathematical Society, Volume 13. (1966)