Arithmetic Formulae and the Use of Subroutines in SAKO

A. W. MAZURKIEWICZ
Institute of Mathematical Machines, Polish Academy of Sciences, Warsaw

SUMMARY—This paper describes the arithmetic formulae and methods of using subroutines in the SAKO automatic programming system. The form of permissible expressions and the basic ideas of how SAKO is translated into machine language are given.

1. INTRODUCTION

ONE aim of a scientific autocode is to assist the programmer to code some class of arithmetic operations. Naturally it is desirable that this class should be as large as possible. It is, however, limited by the 'translator' program.

Among well-known autocodes of this nature are: FORTRAN (1, 2), the Mercury Autocode system (3), and the Univac MATH-MATIC system (4). Each of these has its particular method of writing arithmetic formulae subject to certain limitations. One results from the generally accepted principles of mathematical notation; the second results from the way in which the autocode is used on an actual machine. While developing SAKO for use with arithmetic formulae, we endeavoured:

1. to retain the principles of the mathematical notation as far as possible; and
2. to reduce to a minimum any limitations due to the translator.

In this way it was possible significantly to shorten the time required to program arithmetic operations, and to reduce the possibility of mistakes.

We also endeavoured to incorporate in SAKO all the facilities of the machine language.

Another important feature of an autocode is its method of using subroutines. It should be so constructed that, once written, a subroutine defines a new command of the autocode. Such a system allows one to extend the list of autocode commands and must be elaborated very carefully (3).

2. ARITHMETIC EXPRESSIONS AND FORMULAE

2.1. *Constructing arithmetic expressions*

The scope of SAKO is illustrated by the following examples, all of which are permissible arithmetic expressions in the language:

$A + 1$
$5 \times A + C \times D$
$A +$ ALPHA$/(X1 + 1{\cdot}2345)$
GAMMA $- B3(I,J)$
LOG (SIN (X))$/F(A + B, 45{\cdot}6789)$
$A(3, B(4,J),$ ENT$(B)) + S$
SIGMA$*A$

The symbols $+$, $-$, $\times$, $/$, have their conventional meaning of addition subtraction, multiplication and division. The sign * is used to denote exponentation, thus $A*B$ is the same as A^B.

As usual, parentheses determine the order in which operations are executed or to close the arguments of functions and the subscripts of subscripted variables. Each arithmetical expression has a value uniquely determined by the operations in the expression. This value is either an *integer* or a fractional (real) number.

An arithmetical expression is defined recursively as follows:

(1) A variable or a number is an expression; (2) if A is an expression, then so is (A); (3) if A is an expression which does not begin with the sign $+$ or $-$, then $-A$ is also an expression; (4) if A, B are expressions, and B does not begin with $+$ or $-$, then $A + B$, $A - B$, A/B, $A \times B$, $A*B$, are also expressions; (5) if G is the name of an array, and $A, B, \ldots Z$ are expressions which take integer values, and their number is equal to the dimension of this array, then $G(A, B, \ldots Z)$ is also an expression; (6) if F is the name of a function, and $A, B, \ldots Z$ are expressions

of the same form and number as the arguments of this function, then $F(A, B, \ldots Z)$ is also an expression.

Such a recursive definition enables us to build up an expression by means of superpositions of less complicated expressions; it also enables us to check that actual arithmetic expressions have been correctly formed.

2.2. *Hierarchy of operations*

In the following the operations in SAKO are listed in order of decreasing priority:

(1) Evaluation of functions and subscripted variables; (2) operation *; (3) operation ×; (4) operation /; (5) operation −; (6) operation +.

The 'power' of an operation is defined as follows:

1. Of two operations, that which is enclosed in the greater number of parentheses has the greater power.

2. In the case of an equal number of parentheses that operation which is higher in the above list has the greater power.

3. When two like operations are enclosed in the same number of parentheses, that operation which is to the left of the other has the greater power.

An operation is said to be *evaluable*, when it is followed by any operation of lower power. An arithmetic expression is evaluated in the following way:

1. The first evaluable operation to the left is found, its value computed and this operation replaced by the computed value.

2. This process is repeated until all the operations in the expression are exhausted. This process will be discussed in more detail in the second part of this paper.

2.3. *Integral and fractional expressions*

A SAKO expression can take either an integral or fractional value, or, simply, is either *integer* or *fractional*. The following are the relevant rules:

1. Simple or subscripted variables and functions have integer values if and only if they are declared INTEGER.

2. Function ENT ( ), which computes the integral part of the argument is necessarily integer.

3. If A is integer, then (A) and $-A$ are integer. A similar rule concerns fractional expressions.

4. The expression A/B is always fractional. But $A + B$, $A - B$, $A \times B$, $A*B$ are integer if and only if both A and B are integer.

2.4. *Scale of computation*

This is dealt with in the previous paper (5).

2.5. *Arithmetic formulae*

Arithmetic expressions are mainly used to build arithmetic formulae.

Examples:

ALPHA = BETA + GAMMA
$A = A + 1$
$C(I, J) = A(I, K) \times B(K, J) + C(I, J)$
$B =$ INTEGRAL $(A, B, 0{\cdot}0001, F(\ ))$

In such sentences the value of the expression on the right-hand side of the = sign is computed and is assigned to the variable, simple or subscripted, on the left of this sign. An arithmetic formula determines a value of a variable or changes the previous value of that variable.

In this connexion, we note:

1. If the variable A is integer, and the expression B is fractional, then the formula $A = B$ assigns to A the value of the expression B, rounded up to an integral number.

2. If the variable A is fractional, and the expression B is integer, then the variable A is assigned the value of B reduced to a fraction in conformity with the most recent SET SCALE instruction.

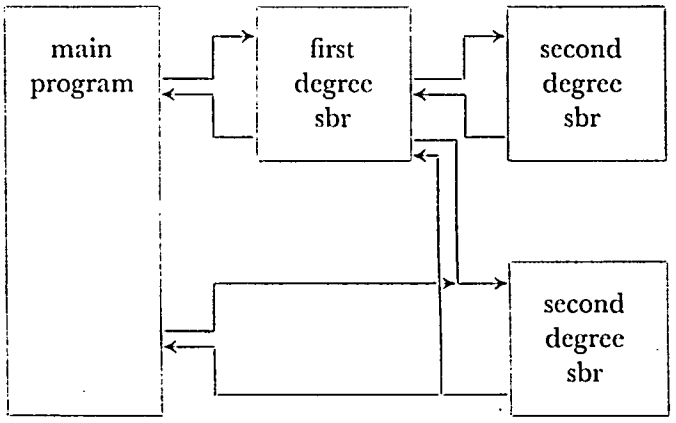
3. SUBROUTINES IN 'SAKO'

3.1. *Terminology and indications*

At this point we explain some of the terms used in what follows. A program calling a subroutine is called the *leading program* in relation to the given subroutine, the subroutine itself being *subordinate* to the leading program. Generally, a subroutine may be called in several times during a computation.

The leading program itself may also be a subroutine. In this case we have a *multilevel* use of subroutines. That part of a program that is not subordinated to any other part is the *main program.* Each chapter of a SAKO program has exactly one main program, and, eventually, a number of subroutines. Subroutines dependent only upon the main program are

first-degree subroutines. Subroutines dependent upon first-degree subroutines are called *second-degree* subroutines, and so on. The way in which a subroutine is used determines the degree of the subroutine.



Multilevel use of subroutines

The action of a subroutine is to obtain a set of results by performing some arithmetic or boolean (see (5)) operations upon a set of arguments:

$$(U, V, \ldots W) = F(X, Y, \ldots Z)$$

For instance, a transformation from one coordinate system to another is such an operation, with the old coordinates as arguments and the new as results.

If the result of an operation is a single number, this operation is called a *function*:

$$U = F(X, Y, \ldots Z)$$

Each subroutine has a strictly determined set of arguments and results, which are expressed by means of *general* symbols. To call in the subroutine these symbols must be replaced by particular symbols which have a concrete meaning. This process of replacing general symbols by particular ones is called *substitution.*

The argument of operation may be some mathematical object as number, vector, complex number, and so on. In particular, an operation (not its value) may also be an argument of another operation. Operations, whose arguments are also operations, must be distinguished.

1. An operation, none of the arguments of which is an operation, is a *first range* operation.

2. An operation, at least one argument of which is an operation of range N, and no other argument is of a higher range, is an operation of *range* $N + 1$.

A subroutine determining an N-range operation is called a subroutine of range N. *Second-range* operations, the result of which is a single number, are called *functionals*. An example of a functional is an integral of a given function. This operation computes the value of integral C, having as arguments the limits of integration, A and B, the number ε (specifying the desired precision of the computation) and the integrand $g(x)$:

$$C = \int_A^B g(x)\,dx + R, \quad |R| < \varepsilon$$

So this operation is of the type:

$$C = F(A, B, \varepsilon, g(\ ))$$

Second- and higher-range operations often arise in computing problems. So we require a system of using subroutines which will enable us, relatively easily, to code higher-range operations.

3.2. *General properties of the subroutines system in SAKO*

The SAKO system is so designed that it is possible:

1. to use subroutines of any range;
2. to obtain a multilevel use of subroutines, limited only by the capacity of the machine storage;
3. to label the instructions and variables in each of the subroutines, as well as in the main program, independently;
4. to substitute arguments into a subroutine by any subroutine of a lower range;
5. to treat subroutines, the result of which is a single number, as language functions;
6. to use as arguments of a subroutine: numbers, variables, arithmetic expressions, arrays and operations;
7. to obtain, as results of a computation, numbers and arrays.

3.3. *Writing subroutines in SAKO*

Each chapter of a program contains its main program and the corresponding subroutines, which are written in standard SAKO. A subroutine always begins with the declaration:

SUBROUTINE: (list of results) = name (list of arguments)

This declaration specifies the name of the subroutine in question and the number, sequence and character of its arguments and results.

Examples of such declarations are:

SUBROUTINE: (U, V) = TRANS $(X, Y,$ ALPHA)
SUBROUTINE: $(*C)$ = MPM $(*A, *B, N, M, L)$
SUBROUTINE: $(*Z)$ = INT STEP $(*Y, H, N, F(\ ))$

Here an array is denoted by preceding the name of the array by the symbol *, while an operation is denoted by its name followed by a pair of parentheses.

All the declarations and actual instructions in a subroutine are preceded by the declaration SUBROUTINE. Only those variables mentioned in the SUBROUTINE declaration and defined in the course of this subroutine may arise in this subroutine. The value of variables named in the list of arguments of a SUBROUTINE declaration are computed by a higher level program. Therefore, the only connexions between leading and subordinate programs are the formulae calling in the subroutine, and the instructions SUBSTITUTE and RETURN.

The symbolism used in a subroutine is independent of that used in other subroutines and in the main program. This means that, in a subroutine, one may use the same names for variables and the same numerical labels for instructions as in other parts of the program, although they may have different meanings. Any such subroutine may be joined to any leading program. A declaration occurring in a leading program is ignored by the subroutine called in. An example of a simple SAKO subroutine is:

SUBROUTINE: (U, V) = TRANS $(X, Y,$ ALPHA)
C = COS (ALPHA)
S = SIN (ALPHA)
$U = X \times C + Y \times S$
$V = -X \times S + Y \times C$
RETURN

If the operation is a function, i.e. has only a single result which is a number, then the method of writing a subroutine for this function is a little different. The SUBROUTINE declaration corresponding to such a subroutine has the following form:

SUBROUTINE: name (list of arguments)

The last executed arithmetic formula of such a subroutine has the form:

name = suitable arithmetic expression

or, in the case of Boolean result:

name ≡ suitable Boolean expression

(see (5)).

3.4. *Use of Subroutines written in SAKO*

When using subroutines it is necessary to perform the appropriate substitutions as well as to call in the appropriate subroutine. This is done by means of an arithmetic formula if the subroutine determines a function, or by means of an instruction:

$$(U, V, \ldots Z) = F(X, Y, \ldots W),$$

sometimes called an 'operation formula'. The operation formula determines the arguments of the subroutine called. The values that result from performing the operation are assigned to the variables to the left-hand side of the = sign. An operation formula is somewhat analogous to an arithmetic one. An arithmetic formula assigns a value to a single variable, while an operation formula assigns values to the elements of a set of variables, or of arrays. The operation formula may be treated as a set of arithmetic formulae, determining the values of the variables named in the list of results.

An example of the way in which the subroutine TRANS is called by means of an operation formula is:

```
—
—
—
—
(X, Y) = TRANS (Y, C + 5·6789, BETA)
—
—
—

SUBROUTINE: (U, V) = TRANS (X, Y, ALPHA)
—
—
—
—
RETURN
```

The order and form of the symbols in the lists of an operation formula must accord with the sequence and form of the symbols named in the SUBROUTINE declaration. The instruction RETURN causes control to be

transferred back to the leading program at the point immediately following that at which the subroutine was called. If the subroutine determines a function, it is called by means of an arithmetic formula, and the name of such a subroutine is treated as the name of the function. However, it is not necessary always to use an arithmetic or operation formula to substitute the actual parameter values for the formal parameters of a subroutine. Some, or, even, all of the arguments, may be assigned by means of the instruction SUBSTITUTE. This instruction is introduced for the purpose of substituting actual for formal arguments, whether a subroutine is called or not; it is important for the multilevel use of subroutines. This independence is necessary so that it is possible to substitute for the formal parameters actual parameters, computed by the main program, in a subroutine called in by the another subroutine, as it takes place, for instance, while computing the integral of a function dependent on several parameters. The form of a SUBSTITUTE instruction is:

SUBSTITUTE: name (list of substitute arguments)

In the list of substitute arguments we write only those arguments which are to be substituted; in those places which correspond to the remaining arguments, we write points; thus

SUBSTITUTE: TRANS (· , · , 12.345678)

The number of points in the list preceding the given argument determines its position in the full list. It is not, therefore, necessary to write points after the last substitute argument. So SUBSTITUTE: TRANS (· , A, ·) may be written SUBSTITUTE: TRANS (· , A)

In an operation calling a subroutine, we only write those arguments which were not previously replaced, since, at the moment of calling, all the arguments of the subroutine must be determined. If all the formal arguments are to be replaced by actual arguments by means of a SUBSTITUTE instruction, then the operation formula may take the form $(U, V) =$ TRANS.

This facility is of considerable value when using second (or higher) range subroutines to operate on subroutines with an undetermined number of parameters ("by-pass" parameters). It is illustrated by the following example:

We want to evaluate the integral

$$\int_0^{\sin\alpha} G(X, \alpha)\, dX + R, \quad |R| < 0.0001$$

for $\alpha = 0(0.0001)1$

The corresponding piece of the SAKO program is:

```
      ---
      --
      -
      SUBSTITUTE: INTEGRAL (0, · , 0.0001, G( ))
      ---
      --
      -
      -
      -
*12)  -
      -
      SUBSTITUTE: G(· , ALPHA)
      -
      -
      -
      -
      A = INTEGRAL (· , SIN (ALPHA))
      PRINT (3 . 7):A
      REPEAT FROM 12 : ALPHA = 0(0.0001)1
      -
      -
      SUBROUTINE: INTEGRAL (A, B, EPSILON, F( ))
      -
      -
      .. ..
      S = S + H × F(A + N × H)
      -
      -
      -
      INTEGRAL = . . .
      -
      RETURN
      SUBROUTINE: G(X, Y)
      -
      -
      -
      -
      G = . . .
      -
      RETURN
```

In this example INTEGRAL ( ) is a functional, while $G(\ )$ is a function. The subroutine INTEGRAL ( ) is a second range subroutine, the subroutine $G(\ )$ is a first range subroutine. The arguments of the subroutine INTEGRAL are: the limits of integration A and B; the computing accuracy EPSILON; and the integrand $G(\ )$. The program $G(\ )$ depends on two arguments: the variable X, with respect to which we want to integrate $G(\ )$; and the variable Y, a parameter.

The main program uses the SUBSTITUTE instruction to assign values to A, EPSILON and the function $F(\ )$ in INTEGRAL ( ) as well as those of the parameter Y in the function $G(\ )$. Then, using an arithmetic formula it calls in the INTEGRAL subroutine assigning to the upper limit of integration the appropriate value of sin (α). At this stage of the computation one argument of the function $G(\ )$ remains undetermined: the variable X of integration. In subroutine INTEGRAL we use subroutine $G(\ )$ as a subroutine of only one argument; this argument is assigned the value of the arithmetic expression $A + N \times H$. So, at the moment of calling in subroutine $G(\ )$ all its arguments are determined. Evaluation of the integral is repeated for the successive values of ALPHA.

This example shows why it is necessary for the substitute process to be independent of a subroutine process, and, in addition, why it must be possible to omit the points. The subroutine computing the integrand is called in by a standard integral evaluation subroutine which assumes that the integrand has only one independent variable. The parameters of the integrand, the number and form of which depend upon the problem in hand, being computed and substituted by the main program. This facility of SAKO holds for any number of parameters and for subroutines of any range.

4. PRINCIPLES OF PROGRAM TRANSLATION

4.1. *General considerations*

The SAKO system, beside its source language, includes two intermediate languages: the SAS language (Symbolic Address System) and the SAB language (Absolute Address System).

In SAS one machine instruction corresponds to one SAS instruction. Addresses of SAS commands may, in general, be symbolic. Likewise, one SAB instruction corresponds to one machine instruction, but addresses of SAB instructions are real.

In both these languages there are, alongside the instructions, so-called *directives*, which do not correspond to machine instructions. They supply information concerning the construction of these languages and the method of translating them into machine code.

A SAKO program is translated into machine code in two phases: 1. SAKO → SAS; and 2. SAS → SAB (or machine code). In the first phase, an instruction in the SAKO program is replaced by a sequence of SAS instructions. A 'dictionary' of instruction labels is also built up, which determines the places of labeled instructions in the machine store. A separate dictionary of instruction labels is constructed for each subroutine. In the second phase, SAS instructions are changed into SAB form by replacing the symbolic addresses by the actual addresses, by means of a simultaneously created 'variable dictionary'.

4.2. *Translation of arithmetic expressions*

To each arithmetic operation corresponds a specific natural number:

=	0	/	3
+	1	×	4
−	2	*	5

There are, in addition, some other operations, for instance the 'indexing operation' IND to which the number 6 corresponds; while to each language function, or function defined by a subroutine, successive integers from 8 to 63 are assigned.

The translation of an arithmetic expression from SAKO to SAS takes place in three phases as in the following example:

Consider the expression: $A \times B + (C(I, J) - A \times B)/(E + D)$

Phase one

The subscripted variables are reduced to one subscript as follows:

$$A(i_1, i_2, \ldots i_n) = A(\ldots((i_1 \times d_2 + i_2) \cdot d_3 + i_3) \ldots + i_n)$$

giving, in the present case

$$C(I, J) = C(I \times N + J)$$

where $d_1, d_2, \ldots d_n$ are the dimension of the array A (N is the second dimension of C). Signs for operations are then replaced by the corresponding code integers; if an operation is in p parentheses, one adds $p \cdot 64$ to the number of the operation. Using octal, we obtain as a result of this first phase of translation:

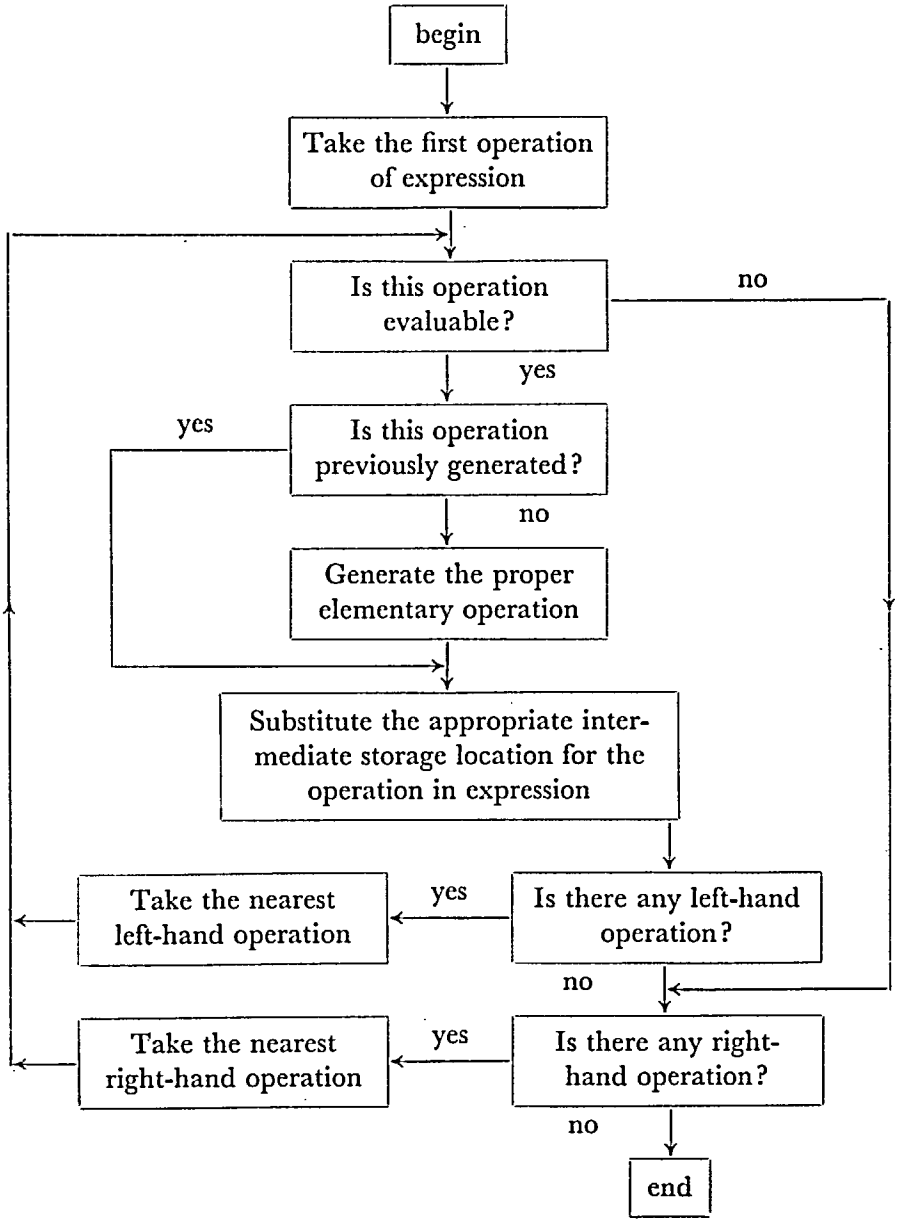
$$A(4)B(1)C(106)I(204)N(201)J(102)A(104)B(3)E(101)D*$$

In this way the process of finding the sequence of what we have called *evaluable operations* is made quite simple: an operation is evaluable if it is followed by another of lower number.

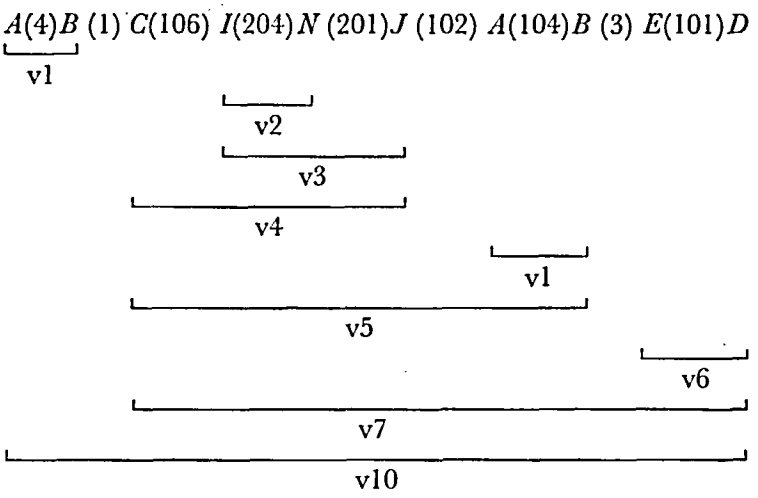
* For the third operation from the left (106), for example, $p = 1$, $Ind = 6$, giving, $1 \cdot 8^2 + 6 = 106$ in octal.—Ed.

Phase two

The expression resulting from the first phase is now processed as follows:



The procedure for the present example can be illustrated in the following manner:



In this way we obtain the following sequence of elementary operations:

$$v1 = A \times B;\ v2 = I \times N;\ v3 = v2 + J;\ v4 = C \text{ Ind } v3;$$
$$v5 = v4 - v1;\ v6 = E + D;\ v7 = v5/v6;\ v10 = v1 + v7$$

Phase three

A sequence of SAS commands for the elementary operations already built is formed and optimized by erasing needless instructions and needless intermediate storage locations, with the following result:

Load *M*	*A*
Multiply	*B*
Left shift	α
Round	
Store Acc	w1
Load *M*	*I*
Multiply	*N*
Left shift	17
Add	*J*
Store Acc	w2
Load *B*	w2
Load Acc	*C* +*
Subtract	w1
Store Acc	w2

* Sign + denotes instructions to be modified.

Load Acc	E
Add	D
Store Acc	w3
Load Acc	w2
Right shift	α
Divide	w3
Store M	w2
Load Acc	w2
Add	w1

Here α is determined by a SET SCALE instruction.

That section of the compiler, which translates arithmetic formulae and expressions occupies nearly half the storage space of the whole SAKO → machine language translator.

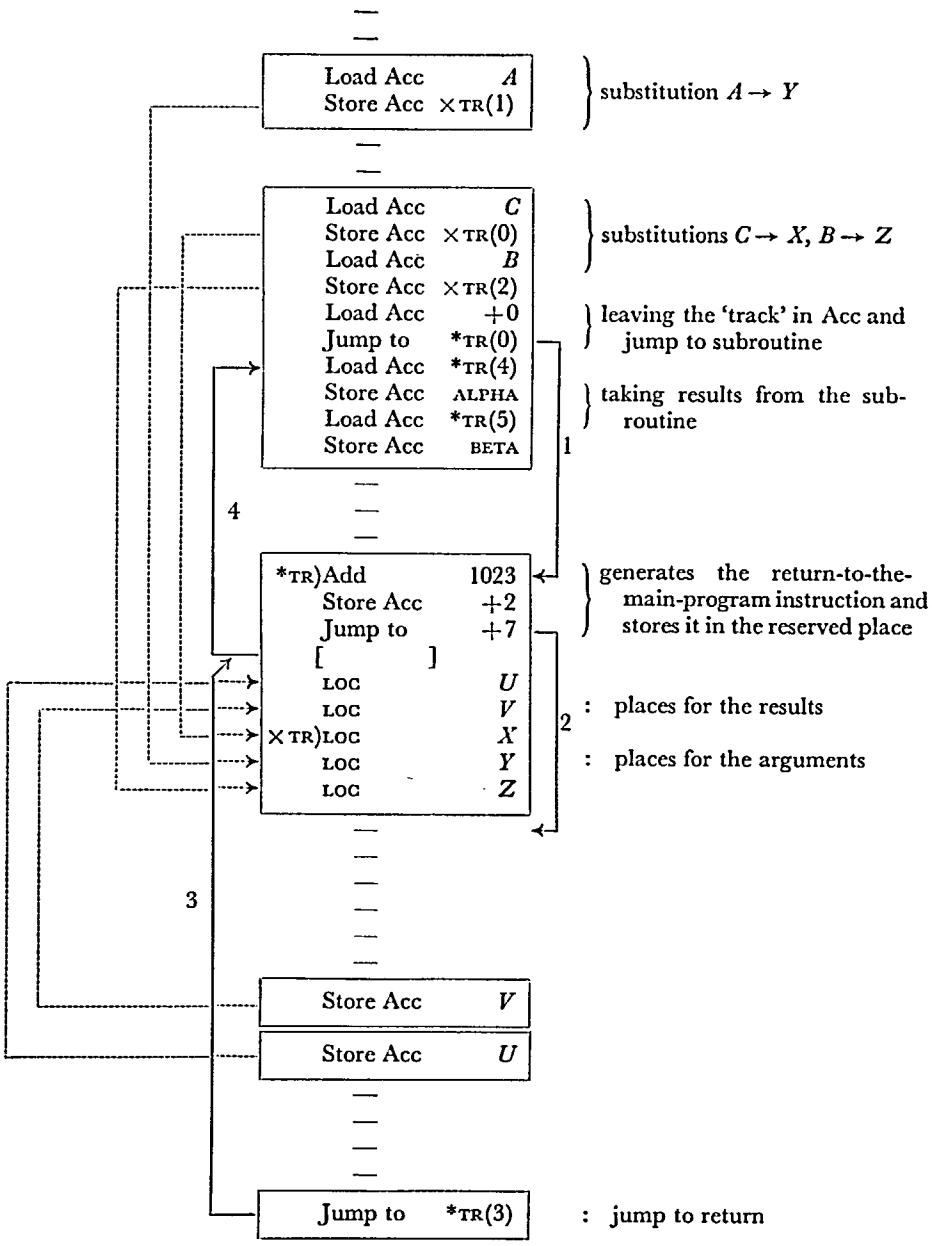
4.3. *Translation of subroutines*

Consider:

```
—
—
SUBSTITUTE: TR (·, A)
—
—
(ALPHA, BETA) = TR(C, ·, B)
—
—

SUBROUTINE: (U, V) = TR(X, Y, Z)
—
—
V = . . .
—
—
—
—
U = . . .
—
—
—
—
—
RETURN
```

When translated into SAS this becomes:



Here continuous lines indicate control transfers, the numbers against them indicating the order in which they are executed. Broken lines indicate the storing of arguments and results in locations reserved for them.

It will be seen that there are two SAS instructions corresponding to the single SAKO instruction SUBSTITUTE. The second stores the value A in its place in the subroutine. The next group of 10 SAS instructions corresponds to the operation formula:

$$(\text{ALPHA, BETA})\tilde{} = \text{TR}(C, \cdot, B)$$

The first four instructions of this group transfer arguments C and B into their places in the subroutine. The next two instructions of this group leave the so-called 'track' in the accumulator (see (6)) and transfer control to the subroutine. The subroutine, completed, re-enters the main program at the next instruction in this group, the first of four instructions which take the results of the subroutine. (The fourth and fifth places of the subroutine TR counting from zero.) Numbers 4 and 5 are computed from the formula: $3 + m$, where m is the ordinal number of the argument. In our case $m = 1, 2$.

The declaration:

$$\text{SUBROUTINE: } (U, V) = \text{TR}(X, Y, Z)$$

is replaced by a group of instructions occupying the nine first places of the subroutine. The first instruction of this group adds a constant to the actual contents of the accumulator, thus generating, in the accumulator, the instruction returning control to the main program. The second instruction writes this return instruction in the third reserved place of the subroutine.

The third instruction transfers control to the subroutine proper, by-passing the places reserved for the subroutine arguments and results. The number +7, the address of this instruction, shows that the jump is to the seventh instruction on. It is computed from the formula: $2 + k + m$, where k is the number of arguments, and m the numbers of results.

The next five places are occupied by SAS 'directives'. They are replaced by free locations during the SAS → SAB translation. LOC 'directives' play a very important role in subroutines. Symbolic addresses U, V, X, Y, Z occurring in instructions correspond in SAB to the locations in which the 'directives' LOC U, LOC V, LOC X, LOC Y, LOC Z are stored. This property of LOC 'directives' is the basis for parameter substitution in subroutines.

The last group, corresponding to the instruction RETURN, consists of

one instruction causing a jump to the return instruction stored in the third (counting from zero) place of the subroutine TR.

Actually, translation of subroutines written in SAKO is, generally, more complicated than the example indicates. For instance, substitution of an operation (or an array) requires that the address, not the value, should be written into the corresponding reserved place of the subroutine. Nevertheless, the basic principles of this translation are the same and are based on the properties of LOC 'directives'.

5. CONCLUSIONS

While developing the SAKO system, we kept in mind the need both to keep the translator as simple as possible and to restrict as little as possible the structure of arithmetic expressions. The principle, 'the result of superposing two arithmetic expressions is an arithmetic expression', was followed as far as possible, but, when the translator threatened to become too complicated, it was decided to accept an object program not fully optimized.

So far as the use of subroutines was concerned, we arrived at a system in which transformations of the type

$$(Y_1, Y_2, \ldots Y_n) = F(X_1, X_2, \ldots X_m),$$

where the X's and Y's may represent different mathematical objects, such as numbers, expressions, vectors, matrices, functions, and so on, were possible. At the same time the possibility of creating a wide range of functionals was realized. That formal arguments should be capable of being replaced by actual arguments, independently of whether or not the operations in question are carried out is considered essential.

The SAKO system, a sketched outline of which is presented in this paper, is not completed. An extension of the source language and an improvement of the translator are both possible and are envisaged. Our main aim is to improve the efficiency of the object program and to extend the scope of the source language by adding more declarations.

ACKNOWLEDGMENTS

The author wishes to acknowledge the contributions of Dr. L. Łukaszewicz, J. Borowiec, M. Lacka, J. Swainiewicz and P. Szorc, who proposed several of the SAKO concepts described above. The author is also extremely grateful to the Editor who was troubled with the proper formal adjustments.

REFERENCES

1. BACKUS, J. W., BEEBER, R. J., BEST, S., GOLDBERG, R., HAIBT, L. M., HERRICK, H. L., NELSON, R. A., SAYRE, D., SHERIDAN, P. B., STERN, H., ZILLER, I., HUGHES,

R. A. and NUTT, R., 'The FORTRAN automatic coding system'. *Proc. of the Western Joint Computer Conference*, p. 188 (February, 1957).
2. I.B.M., 'FORTRAN automatic coding system', Reference Manual (1958).
3. BROOKER, R. A., RICHARDS, B., BERG, E., KERR, R. H., *The Manchester MERCURY Autocode System*. Computing Machine Laboratory, the University, Manchester (May, 1959).
4. 'UNIVAC Math-Matic Programming System', Reference Manual, Sperry Rand Corporation (1958).
5. ŁUKASZEWICZ, L., 'SAKO—An Automatic Coding System'. This volume, p. 161 (1961).
6. MACCRACKEN, D. D., *Digital Computer Programming*. John Wiley (1957).