The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System

PETER B. SHERIDAN, *International Business Machines Corporation, New York, N. Y.*

1. Introduction

The present paper describes, in formal terms, the steps in translation employed by the FORTRAN arithmetic translator in converting FORTRAN formulas into 704 assembly code. The steps are described in about the order in which they are actually taken during translation. Although sections 2 and 3 give a formal description of the FORTRAN source language, insofar as arithmetic type statements are concerned, the reader is expected to be familiar with FORTRAN II, as well as with SAP II and the programming logic of the 704 computer.

The first major phase of translation, described in sections 4–7, is concerned with converting an arithmetic formula, regarded as the statement of an algorithm, into a set of triples $(C_i, \Diamond_i, N_i)$ which also describe the algorithm, but in a manner which lends itself more easily to conversion into conventional computer code (not necessarily that of the 704). The three elements of a triple have essentially the following meanings:

$\Diamond_i$: operation to be performed
N_i : operand
C_i : "time" at which the operation must be performed.

There are as many distinct "times" of operation as there are subexpressions in the entire expression to be evaluated. For example, consider the arithmetic expression[1]

$$(((A + B) - C)/((D * (E + F)/G) - H + J)). \qquad (1)$$

One may regard this expression as containing as many subexpressions as there are distinct pairs of parentheses shown—namely six. To evaluate the entire expression all six subexpressions must be evaluated. While there is some latitude in the order in which these subexpressions are to be evaluated, not all orders will work (since "higher level" subexpressions depend on "lower level" ones for their values). A possible order of evaluation for the example shown is as follows:

1. $(A + B)$
2. $((A + B) - C)$
3. $(E + F)$
4. $(D * (E + F)/G)$
5. $(D * (E + F)/G) - H + J$
6. $(((A + B) - C)/((D * (E + F)/G) - H + J))$.

The triples representing this computation would break up into six subsets, called *segments* according to their C-number (first member of a triple). In each segment (corresponding to a subexpression) there are as many triples as there are terms in the subexpression. (The number of triples in a segment may be called the *length* of a segment.) Using the numbers written alongside the subexpressions as names for these subexpressions, we get the following representation of the computation in triples notation:

$$\begin{aligned}&(1, +, A)(1, +, B)(2, +, 1)(2, -, C)\\ &\qquad\qquad(3, +, E)(3, +, F)(4, *, D)\\ &(4, *, 3)(4, /, G)(5, +, 4)(5, -, H)\\ &\qquad\qquad(5, +, J)(6, *, 2)(6, /, 5)\end{aligned} \qquad (2)$$

If the divide operation is regarded as multiplication by the inverse, then it is clear that the triples belonging to any given segment of (2) could be reshuffled into some different order without disturbing the algorithm. Although not all possible segments which the FORTRAN translator produces allow reshuffling, the latitude, whenever present is later used to obtain economies in the computer code eventually produced. It should also be observed that since each triple bears its own segment number, extensive rearrangements of the triples could be tolerated without making it impossible to restore them into proper order. The FORTRAN translator takes advantage of this fact in that it does not necessarily produce the triples segment by segment with segments in proper order. However, the deviations from this ultimately required order which the process of triples generation introduces are correctible by reordering. The segment numbers developed by the FORTRAN translator are in reverse numerical order, i.e., the largest segment number represents the subexpression first to be evaluated, while the smallest segment number represents the last (which always coincides with the entire expression). The first major objective of the FORTRAN translator is accomplished in a number of steps and consists in the translation of form (1) into form (2).

The first step, described in section 4, is to replace constants and subscripted variables by simple variables, thus ensuring that all of the arguments entering into the computation are of a uniform nature.

The next step, described in section 5, is aimed at ensuring that every subexpression in the expression to be evaluated is provided with an explicit pair of parentheses. In writing FORTRAN formulas one need not indicate all parentheses, the usual order of precedence among arithmetic operations being assumed: exponentiation, multiplication and division, addition and subtraction. The method by which these precedence relations are made explicit by the FORTRAN translator is roughly the following: An arithmetic connective of high precedence may be

[1] In the FORTRAN system * is used for multiplication, ** for exponentiation.

Algorithm for Computing Triples from Normal Forms (Example)

(Read from left to right)

(Column in which the arrowhead of an operation symbol appears is the "time" at which the operation is executed; more than one operation may be performed at a time.)

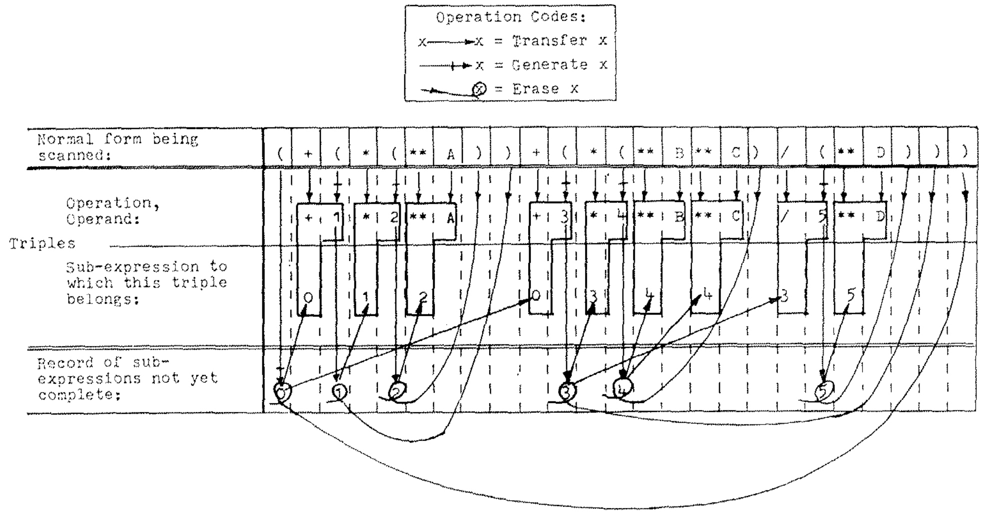

Fig. 1.

thought of as weakly separating the arguments to either side of it, while a connective of low priority strongly separates the arguments to either side of it. Since there are three degrees of precedence to be considered, we may represent three degrees of separation power by placing one, two, or three pairs of parentheses to either side of a connective according to its order of precedence, thus:

	$A ** B$	$A) ** (B$	(exponentiation)
convert	$A * B$ into	$A)) * ((B$	(multiplication)
	$A + B$	$A))) + (((B$	(addition)

This introduction of parentheses is *balanced* in the sense that as many left parentheses are introduced as right ones. If, in addition to inserting parentheses as shown above, one also prefixes the entire expression by three left parentheses and closes it by three right parentheses, then—as the reader may convince himself—a correct parenthesization of the expression is accomplished even though many unnecessary (though harmless) pairs of parentheses are also inserted. For example:

$$A + B ** C/D \tag{3}$$

becomes

$$(((A))) + (((B) ** (C))/((D)))$$

In the FORTRAN translation four rather than three levels of precedence are recognized because the character "," which conventionally separates successive arguments of a function—as in $f(a, b, c)$—is also treated as an arithmetic connective. Also, for reasons having to do with the subsequent generation of triples, each inserted left parenthesis is preceded by an operation sign which essentially tags the parenthesis with its *strength*:

$$+(*(**(A))) + (*(**(B) ** (C))/(**(D))) \tag{4}$$

Arithmetic expressions expanded by the insertion of parentheses and operation signs as indicated above (and carefully described in section 5) are called arithmetic expressions in *normal form.*

The next step in translation, described in section 6, is the generation of a list of triples which (except for the inclusion of some extraneous list members) constitutes an unsorted representation of the desired algorithm. A single scan forward on expressions of a form equivalent to that shown in (3) above produces the desired triples. Every left parenthesis marks the beginning of a subexpression; the operation sign preceding it shows in what sort of arithmetic operation that subexpression is to participate as a term. A running index, N_i, (N for *next subexpression*), which steps up one every time another left parenthesis is crossed, serves to generate a set of subexpression numbers which become the indices N_i of the generated triples. The numbers C_i (C for *current subexpression*) may be generated from N_i provided that one keeps a record of what subexpression numbers were generated at previous left parenthesis crossings, and steps back to the proper earlier subexpression after having crossed one or more right parentheses.

This procedure is illustrated in figure 1, using expression (4) as a source of triples production.

Section 7 describes how to eliminate extraneous triples from lists described as above. To begin with, one can (with only one exception) eliminate all triples which form a single segment by themselves. Secondly, after sorting the triples into order, one can eliminate duplication between entire segments of triples which can be recognized to be identical sub-expressions.

Section 8 describes some intra-segment rearrangements of triples which lead to an ultimate order of computer operation which is not randomly wasteful of storage-to-register and register-to-storage transfers in the course of expression evaluation.

When once an expression has been converted into a string of triples, properly rearranged and made as short as possible by elimination of duplication, it is ready for conversion into one-address SAP assembly code. In most triples the last element N_i will have to be interpreted as an SAP address of a number to be operated upon by an operation corresponding to $\Diamond_i$.

Sections 9 and 10 show how the FORTRAN translator assigns storage locations and computes addresses for the various classes of operands encountered in these triples. The following classes of operand storage are considered in order:

Integer and floating point constant storage

Input and output arrays (Represented by source language references to subscripted variables)

Working storage for each expression (Represented in triples code by references in one segment to the result produced by another segment of the same expression)

Argument storage for functions

Return jump address storage required when functions are nested and control must pass down a chain and then back up again.

Section 11 actually shows, almost in flow-chart form, how SAP assembly instructions are produced from the strings of triples into which FORTRAN expressions have been recoded.

The Appendix shows how FORTRAN statement compilation can be extended to the case in which the operations consist in Boolean *and, or* and *not.*

The author is indebted to Messrs J. W. Backus, R. A. Nelson, and I. Ziller of IBM for initially suggesting to him the basic ideas underlying section 6 below, and he is most especially grateful to Mr. A. W. Holt of Remington Rand for the extended introduction to this paper, without which its readability would have suffered considerably.

2. Definitions

(*a*) The *alphabet* of FORTRAN comprises the alphameric characters A, $\cdots$, Z, 0, 1, $\cdots$, 9 and the set of "special" characters: •, (,), ,, $+$, $-$, $*$, $/$, $=$, and $\dashv$. The last is not a character explicitly indicated in any FORTRAN statement, serving solely as a statement endmark on the executive level, with which we shall here be primarily concerned. Lower case Greek letters will be used throughout to denote arbitrary characters of the FORTRAN alphabet.

By the term *string* we shall mean a sequence of juxtaposed alphabet characters, of finite length. Upper case Greek letters will be used to denote arbitrary strings.

The *length* $L(\Sigma)$ of a string Σ is the number of literal occurrences of character tokens in its construction. The symbol Λ will be used to denote the null string, of length 0; of course, for any string Σ, $\Sigma\Lambda = \Lambda\Sigma = \Sigma$.

Two strings $\Phi = \varphi_1 \cdots \varphi_m$, $\Psi = \psi_1 \cdots \psi_n$ are said to be *identical* (and we write $\Phi = \Psi$) if and only if $m = n$ and for each i, $\varphi_i = \psi_i$.

A string Φ is said to be *included in* (or to be *a substring of*) a string Ψ (and we write: $\Phi \subseteq \Psi$) if Ψ has the representation $\Psi = \mathrm{A}\Phi\mathrm{B}$ where, A, B are strings. As usual, A is said to be a *head* of Ψ, and B a *tail* of Ψ.

If Ψ is an arbitrary string, then the notation $H_n(\Psi)$, $T_n(\Psi)$, where n is an arbitrary non-negative integer, is defined as follows: If $\Psi = \Phi\mathrm{X}$, then

$$H_n(\Psi) = \begin{cases} \Phi \text{ if } n \leqq L(\Psi) \text{ and } L(\Phi) = n, \\ \qquad\qquad \text{or } n > L(\Psi) \text{ and } \Psi = \Phi \\ \Lambda \text{ if } n = 0 \end{cases}$$

$$T_n(\Psi) = \begin{cases} \mathrm{X} \text{ if } n \leqq L(\Psi) \text{ and } L(\mathrm{X}) = n, \\ \qquad\qquad \text{or } n > L(\Psi) \text{ and } \Psi = \mathrm{X} \\ \Lambda \text{ if } n = 0 \end{cases}$$

Thus, if $\Psi = ABC + D/E ** F$ then

$$H_4(\Psi) = ABC+$$

$$T_5(\Psi) = /E ** F$$

$$H_{17}(\Psi) = \Psi$$

$$T_{23}(\Psi) = \Psi$$

(*b*) *Names* are strings classified as follows:

integer constants

floating-point constants

integer variables

floating-point variables

function names

An *integer constant* is a string $\mathrm{K} = \kappa_1 \cdots \kappa_n$ such that each κ_i is an integer between 0 and 9 (inclusively) and the relation $\mathrm{K} < 2^{15}$ holds.

A *floating-point constant* is a string $\Gamma = \gamma_1 \cdots \gamma_n$ where $\gamma_j = \bullet$ for some j and each γ_i, for $i \neq j$, is between 0 and 9 (inclusively), provided $\Gamma = 0$ or $10^{-38} < \Gamma < 10^{38}$.

An *integer variable* is a string $\mathrm{I} = \iota_1 \cdots \iota_n$ where each ι_j is alphameric, but ι_1 is one of the characters I, J, K, L, M or N, and $L(\mathrm{I}) \leqq 6$.

A *floating-point variable* is a string $\Gamma = \gamma_1 \cdots \gamma_n$ where each γ_i is alphameric and γ_1 is alphabetic but not one of the characters I, J, K, L, M or N. Again, $L(\Gamma) \leqq 6$.

A *function name* is a string $\Phi = \varphi_1, \cdots, \varphi_n$, where each φ_i is alphameric, φ_1 is alphabetic and such that either

(*i*) $L(\Phi) \leqq 6$, Φ does not appear in a DIMENSION[2] statement, and $L(\Phi) < 4$ or $\varphi_n \neq$ F

or (*ii*) $4 \leqq L(\Phi) \leqq 7$ and $\varphi_n =$ F.

Functions denoted by names in the first category are referred to as FS-type functions, and are defined by FORTRAN II FUNCTION or SUBROUTINE subprograms. Those denoted by names in the second category are referred to as FN-type functions, and are defined by machine-coded library tape subprograms, built-in open subroutines, or by a single arithmetic statement (see function definition below). An FS-type function is integer-valued if and only if φ_1 is one of the characters I, J, K, L, M or N. An FN-type function is integer-valued if and

[2] For definition, see, e.g., the FORTRAN Programmer's Reference Manual.

only if $\varphi_1 = \text{X}$. (This discrepancy in notational convention is, in principle, a nonessential one and arose only through historical accident.)

(c) A *subscript* is a string of one of the following forms:

$$\Sigma$$

$$\text{K}$$

$$\text{K} * \Sigma$$

$$\Sigma + \text{K}$$

$$\Sigma - \text{K}$$

$$\text{K} * \Sigma + \text{P}$$

$$\text{K} * \Sigma - \text{P}$$

where K, P are integer constants and Σ is an integer variable. In addition, the magnitude of a subscript cannot exceed $2^{15} - 1$.

(d) A *subscripted variable* is a string of the form

$$T(\Sigma_1, \cdots, \Sigma_k)$$

where T is an integer or floating-point variable and such that if $T = \tau_1 \cdots \tau_n$,

(*i*) T appears in a DIMENSION statement

and (*ii*) $L(T) < 4$ or $\tau_n \neq \text{F}$

and the Σ_i are subscripts with $k \leqq 3$.

3. Rules of Formation

(I) The set of *expressions* is recursively characterized as follows:

*E*1. A constant or variable name Φ is an expression of the same mode as Φ. If $\Phi = \varphi_1 \cdots \varphi_n$ and $L(\Phi) < 4$ or $\varphi_n \neq \text{F}$, and Φ appears in a DIMENSION statement, and if, furthermore, $\Sigma_1, \cdots, \Sigma_k$, $(1 \leqq k \leqq 3)$, are subscripts, then $\Phi(\Sigma_1, \cdots, \Sigma_k)$ is an expression of the same mode as Φ.

*E*2. If Φ is an expression not of the form $+\Psi$ or $-\Psi$, then $+\Phi$ and $-\Phi$ are expressions of the same mode as Φ.

*E*3. If Φ is an expression, then (Φ) is an expression of the same mode as Φ.

*E*4. If Φ is an n-adic function name and $A_1, \cdots, A_n$ are expressions, then $\Phi(A_1, \cdots, A_n)$ is an expression of the same mode as Φ.

*E*5. If Φ, Ψ are expressions of the same mode and Ψ is not of the form $+\text{X}$ or $-\text{X}$, then $\Phi + \Psi$, $\Phi - \Psi$, $\Phi * \Psi$, Φ/Ψ are expressions of the same mode as Φ.

*E*6. If Φ, Ψ are expressions and Ψ is floating-point mode only when Φ is, then unless either Φ or Ψ is of the form $A ** B$ (A, B variables, constants, or function expressions), $\Phi ** \Psi$ is an expression of the same mode as Φ.

Note that *E*1–*E*6 prohibit, by implication, the writing of mixed-mode expressions except under certain special formal conditions. To wit, the only allowable expressions in our language are those we shall term *integer* and *floating-point* expressions. An integer expression is one in which any floating-point mode name occurs, if at all, within the "argument vector" of some function expression,—for example:

$$\text{I} + \text{XSINF}(\text{A} * \text{B}/\text{C})$$

A floating-point expression, on the other hand, is one in which any integer-mode name occurs, if at all, within the argument vector of a function expression, within the exponent of an exponential, or within a subscript,—for example:

$\text{A}(\text{I}) ** \text{J} + \text{SINF}(\text{K} * \text{L}/\text{M})$

Any other string is referred to as *mixed*, and cannot belong to the set of allowed expressions.

(II) An *arithmetic expression* is a string of the form

$$= \Phi \dashv$$

where Φ is an expression in the sense of *E*1–*E*6.

(III) A *pure arithmetic statement* is a string of the form

$$\Psi = \Phi \dashv$$

where Ψ is a subscripted or nonsubscripted variable, and $= \Phi \dashv$ is an arithmetic expression.

(IV) A *quasi-arithmetic statement* is a string of one of the following forms:

(*a*) $\text{IF}(\Phi)$, where Φ is an expression.

(*b*) $\text{CALL}\ \Phi(A_1, \cdots, A_n)$, where Φ is a function name such that

(*i*) if $\Phi = \varphi_1 \cdots \varphi_n$, then $L(\Phi) < 4$ or $\varphi_n \neq \text{F}$, $L(\Phi) \leqq 6$ and Φ does not appear in a DIMENSION statement

(*ii*) each A_i is an expression, in the sense of *E*1–*E*6 or a Hollerith field (q.v., FORTRAN Programmer's Reference Manual).

The reason for referring to the above as *quasi*-arithmetic statements will be made clear later.

(V) A *function definition* is a string of the form

$$\Phi(A_1, \cdots, A_n) = E[A_1, \cdots, A_n] \dashv$$

where Φ is a function name such that

(*i*) if $\Phi = \varphi_1 \cdots \varphi_n$, then $4 \leqq L(\Phi) \leqq 7$ and $\varphi_n = \text{F}$,

(*ii*) each A_i is a nonsubscripted variable name,

(*iii*) $E[A_1, \cdots, A_n]$ is an expression (in the sense of *E*1–*E*6) in the (free) variables $A_1, \cdots, A_n$. We note in passing, that any A_i may occur vacuously in $E[A_1, \cdots, A_n]$. Thus, $E[A_1, \cdots, A_n]$ is a function form in the free variables $A_1, \cdots, A_n$.

4. Reduction of Expressions

An expression Φ is *reduced* in the following manner:

(I) Each integer or floating-point constant occurring in Φ, but not contained in a subscript, is replaced by an integer or floating-point variable name, respectively, provided the replacement is made at all occurrences of that constant and the replacement name does not already occur in Φ.

(II) Each integer or floating-point subscripted variable is replaced by a nonsubscripted integer or floating-point variable name, respectively, in such a manner that the replacement name does not already occur in Φ and such that the replacement of each subscripted variable name is made at all occurrences of the latter in Φ. Also, no subscripted variable can have the same name as a nonsubscripted variable, nor can any two n-dimensional subscripted variables have the same names if their first $n-1$ dimensions are respectively identical.

The result, then, of applying procedures (I), (II), above, to the expression Φ will be termed its *reduced* form and denoted by Φ^R.

Now, if Φ is an expression, then we shall denote by $\mathfrak{B}_\Phi$ the set of all non-null connected name strings occurring in Φ. Thus, for example, if

$$\Phi = \text{ABC} * (-\text{XY})$$

then $\mathfrak{B}_\Phi = \{\text{A, B, C, AB, BC, ABC, X, Y, XY}\}$. Clearly, for any expression Φ, there exists at least one element $\Sigma \in \mathfrak{B}_\Phi$ such that if $\Psi \in \mathfrak{B}_\Phi$ and $\Sigma \subseteq \Psi$, then $\Sigma = \Psi$, i.e., Σ is, in this sense, a *maximal* element. We shall denote by $\mathfrak{W}_\Phi$ the subset of $\mathfrak{B}_\Phi$ consisting of its maximal connected non-null name strings. In the last example, $\mathfrak{W}_\Phi = \{\text{ABC, XY}\}$.

5. Normal Form of an Arithmetic Expression

In this section we shall define a set of transformations which, beginning with any reduced arithmetic expression $= \Phi^R \dashv$, recursively generate what we shall call *partial normal forms* $\Delta_i X$ where $T_1(X) = \dashv$ and Δ_0 is the character $=$. The recursion on the head-strings Δ_i is as follows:

If $\qquad \Delta_i \Psi X \to \Delta_i \Psi' X,$

then $\qquad \Delta_{i+1} = \Delta_i \Psi'.$

We shall assume throughout that $\Sigma \in \mathfrak{W}_\Phi \cup \{(\}$.

$T1.\ \Delta_i \Sigma X \to \Delta_i + (*(**(\oplus \Sigma X \qquad \text{if } T_1(\Delta_i) \in \{=, (\}$

$T2.\ \Delta_i \pm \Sigma X \to \begin{cases} \Delta_i \pm (*(**(\oplus \Sigma X & \text{if } T_1(\Delta_i) \in \{=, (\} \\ \Delta_i\ )\ )\ ) \pm (*(**(\oplus \Sigma X, & \text{otherwise} \end{cases}$

$T3.\ \Delta_i * / \Sigma X \to \Delta_i\ )\ ) * /(**(\oplus \Sigma X$

$T4.\ \Delta_i ** \Sigma X \to \Delta_i\ ) ** (\oplus \Sigma X$

$T5.\ \Delta_i (X \to \Delta_i \oplus (X \qquad \text{if } T_1(\Delta_i) \in \mathfrak{B}_\Phi$

$T6.\ \Delta_i) X \to \Delta_i\ )\ )\ )\ ) X$

$T7.\ \Delta_i , X \to \Delta_i\ )\ )\ )\ ) \oplus (X$

$T8.\ \Delta_i \dashv \to \Delta_i\ )\ )\ )$

Note that the recursion is terminated by $T8$, since $X = \dashv$ at this step. Note, further, that stratification (by levels) of the arithmetic expression $= \Phi \dashv$ via the above schema proceeds from the assumption that the order (or degree) of "binding" of operations appearing in Φ coincides with that of conventional mathematical usage, viz. (in order of increasing "strength"): $\pm$; $*/$; $**$; and $\oplus$. Of course, the last noted "operation" is associated solely in each partial normal form with functions and has only a syntactical purpose in that context. Thus, $T1$–$T8$ reflect and render more explicit normally assumed usage regarding stratification of algebraic expressions.

If we denote by $N_i(\Phi)$ the ith partial normal form of Φ, and by $N(\Phi)$ the last of these (which we may refer to simply as the *normal form* of Φ), then the following example should suffice to illustrate the above schema.

Let $\Phi = -\text{XYZF}(\text{A},\ \ \text{B} * \text{C} ** (-\text{D}))/\text{E} + \text{F}$. Then,

$N_0(\Phi) = \Delta_0 \Phi \dashv$, with Δ_0 standing for $=$

$N_1(\Phi) = \Delta_1(\text{A, B} * \text{C} ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_1 = \Delta_0 - (*(**(\oplus \text{XYZF}$

$N_2(\Phi) = \Delta_2 \text{A, B} * \text{C} ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_2 = \Delta_1 \oplus ($

$N_3(\Phi) = \Delta_3 , \text{B} * \text{C} ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_3 = \Delta_2 + (*(**(\oplus \text{A}$

$N_4(\Phi) = \Delta_4 \text{B} * \text{C} ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_4 = \Delta_3\ )\ )\ )\ ) \oplus ($

$N_5(\Phi) = \Delta_5 * \text{C} ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_5 = \Delta_4 + (*(**(\oplus \text{B}$

$N_6(\Phi) = \Delta_6 ** (-\text{D}))/\text{E} + \text{F} \dashv$,
with $\Delta_6 = \Delta_5\ ))*(**(\oplus \text{C}$

$N_7(\Phi) = \Delta_7 - \text{D}))/\text{E} + \text{F} \dashv$, with $\Delta_7 = \Delta_6\ )**(\oplus($

$N_8(\Phi) = \Delta_8))/\text{E} + \text{F} \dashv$, with $\Delta_8 = \Delta_7 - (*(**(\oplus \text{D}$

$N_9(\Phi) = \Delta_9)/\text{E} + \text{F} \dashv$, with $\Delta_9 = \Delta_8\ )\ )\ )\ )$

$N_{10}(\Phi) = \Delta_{10}/\text{E} + \text{F} \dashv$, with $\Delta_{10} = \Delta_9\ )\ )\ )\ )$

$N_{11}(\Phi) = \Delta_{11} + \text{F} \dashv$, with $\Delta_{11} = \Delta_{10}\ ))/(**(\oplus \text{E}$

$N_{12}(\Phi) = \Delta_{12} \dashv$, with $\Delta_{12} = \Delta_{11}\ )\ )\ ) + (*(**(\oplus \text{F}$

and, finally,

$$N(\Phi) = N_{13}(\Phi) = \Delta_{13}, \quad \text{with } \Delta_{13} = \Delta_{12}\ )\ )\ ).$$

More explicitly, then, $N(\Phi)$ is the expression

$$= -(*(**(\oplus \text{XYZF} \oplus (+(*(**(\oplus \text{A}\ )\ )\ )\ ) \oplus (\ + (*(** (\oplus \text{B}\ ))*(**(\oplus \text{C}\ )**(\oplus(-(*(**(\oplus \text{D}\ )\ )\ )\ )\ )\ )\ )\ )\ )\ )/ (**(\oplus \text{E}\ )\ )\ ) + (*(**(\oplus \text{F}\ )\ )$$

It may be seen quite easily that the result $N(\Phi)$ of application of $T1$–$T8$ to a reduced arithmetic expression $= \Phi^R \dashv$ is such that the balance of left and right parentheses is not disturbed (*closure* condition). To wit: the three additional left parentheses generated by $T1$ are closed by the three additional right parentheses generated by $T8$, if $T1(\Delta_i)$ is $=$, or $T6$, if $T1(\Delta_i)$ is $($; the same is true for the first line of $T2$, and the second line of $T2$ is self closing; $T3$, $T4$ and $T5$ are self-closing; $T6$ introduces three additional right parentheses which are closed by the three additional left parentheses generated by $T1$ or the first line of $T2$; identical assertions hold for $T7$ and $T8$.

If $N(\Phi)$ is $=\Psi$, then we shall define Φ^N as Ψ, i.e., the string $N(\Phi)$ minus the FORTRAN $=$ sign. Thus, Φ^N is a string of elements of the form $\Diamond_i \Psi_i$ where, for each i

$$\Diamond_i = \Lambda \text{ and } \Psi_i =)$$

or $\qquad \Diamond_i \in \{+, -, *, **, \oplus\}$ and $\Psi_i \in \mathfrak{W}_\Phi \cup \{(\}$.

6. Level Analysis

The level analysis of an arithmetic expression $= \Phi$ consists in the recursive generation of what we shall call *partial productions* Π_i, each partial production a string of triples (of entities to be described below) formed in the following manner.

We define three integer sequences $\{N_i\}$, $\{C_i\}$, $\{A_i\}$ and a sequence $\{K_i\}$ of integer sequences such that, initially, $N_1 = 1$, $C_1 = A_1 = 0$ and $K_1 = \Lambda$. By K_i' we shall mean the last term of the sequence K_i, and if $K_i = (\Theta, K_i')$ then $\bar{K}_i = (\Theta)$ (possibly null). We set the

TABLE 1

	−(	*(	**(	⊕XYZF	⊕(	+(	*(	**(	⊕A	)	)	)	)	⊕(	+(	*(	**(	⊕B	)	)	*(	**(	⊕C	)	**(	⊕(	−(
N_i	1	2	3	4	4	5	6	7	8	8	8	8	8	8	9	10	11	12	12	12	12	13	14	14	14	15	16
C_i	0	1	2	3	3	4	5	6	7	7	6	5	4	3	8	9	10	11	11	10	9	12	13	13	12	14	15
A_i	0	1	2	3	3	4	5	6	7	7	6	5	4	3	4	5	6	7	7	6	5	6	7	7	6	7	8
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

	*(	**(	⊕D	)	)	)	)	)	)	)	)	)	)	/(	**(	⊕E	)	)	)	+(	*(	**(	⊕F	)	)	)	
N_i	17	18	19	19	19	19	19	19	19	19	19	19	19	19	20	21	21	21	21	21	22	23	24	24	24	24	24
C_i	16	17	18	18	17	16	15	14	12	9	8	3	2	1	19	20	20	19	1	0	21	22	23	23	22	21	0
A_i	9	10	11	11	10	9	8	7	6	5	4	3	2	1	2	3	3	2	1	0	1	2	3	3	2	1	0
i	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54

initial partial production $\Pi_1 = \Lambda$, and if $\Pi_i \rightarrow \Pi_i E$, then $\Pi_{i+1} = \Pi_i E$.

Now if $\diamondsuit_i \Psi_i$ denotes the ith element of Φ^N, the production schema is as follows:

$$\text{If } \Psi_i = \begin{cases} (& \text{then } \Pi_i \rightarrow \Pi_i(C_i, \diamondsuit_i, N_i) \\ & N_{i+1} = N_i + 1 \\ & C_{i+1} = N_i \\ & A_{i+1} = A_i + 1 \\ & K_{i+1} = (K_i, C_i) \\ \Sigma, & \text{where } \Sigma \in \mathfrak{W}_\Phi, \\ & \text{then } \Pi_i \rightarrow \Pi_i(C_i, \diamondsuit_i, \Sigma) \\ & N_{i+1} = N_i \\ & C_{i+1} = C_i \\ & A_{i+1} = A_i \\ & K_{i+1} = K_i \\ ) & \text{then } \Pi_i \rightarrow \Pi_i \\ & N_{i+1} = N_i \\ & C_{i+1} = K_i' \\ & A_{i+1} = A_i - 1 \\ & K_{i+1} = \bar{K}_i \end{cases}$$

The effect of applying these transformations completely to the normal form Φ^N is to produce a string $\Pi(\Phi)$—which we shall refer to simply as the *production* of Φ—which is a representation of the computation required to evaluate the original arithmetic expression $= \Phi \dashv$. The computation proceeds from the innermost to the outermost levels (in the sense of $T1$–$T8$) in a systematic manner, as we shall later see.

Note that, since $A_1 = 0$ and

$$A_{i+1} = \begin{cases} A_1 + 1 & \text{when } \Psi_i = (\\ A_i & \text{when } \Psi_i \in \mathfrak{W}_\Phi \\ A_i - 1 & \text{when } \Psi_i =), \end{cases}$$

then the last term of the sequence $\{A_i\}$ is 0 if and only if Φ itself is closed with respect to parenthesization.

Now, each element C (respectively N) of the set of terms of the sequence $\{C_i\}$ (respectively $\{N_i\}$) may be interpreted as the index of a "currently defined" (respectively, "next-to-be-defined") expression S_c (respectively S_N) embedded in the production $\Pi(\Phi)$. Each such expression is defined as the set of all triples of the form $(C, \diamondsuit_c{}^j, \Psi_c{}^j)$, for arbitrary, fixed C, where the index j ranges over the number of triples having the "current" index C.

From the above "partial production" schema, it is easily seen that, for any i, j, $C_i = C_j$ entails $A_i = A_j$ (but not conversely). Thus, since each element of the sequence $\{A_i\}$ is a level index (rising by 1 for each left parenthesis, dropping by 1 for each right parenthesis, and otherwise stationary), it is meaningful to state that each of the above mentioned triples belongs to the same level. We shall call such a set of triples a *segment* of the level to which any of its members belong. Of two segments Sc_i, Sc_j, moreover, we may say that they *belong to* (or *are segments of*) the same level if and only if $A_i = A_j$. Since this is an equivalence relation between segments, we see that each level is completely defined by the set of all segments *belonging* to it, in the above defined sense. Furthermore, since all triples comprising a segment belong to the same level, their operation elements are of the same type, i.e., all $*$ and/or $/$; all $+$ and/or $-$; all $**$; or all $\oplus$.

Before proceeding further, let us pause to illustrate the partial production schema with respect to the example

$$\Phi = -\text{XYZF}(\text{A}, \text{B} * \text{C} ** (-\text{D}))/\text{E} + \text{F}$$

of section 5. To this end, it is convenient to arrange all information in the form of table I, wherein the ith column contains at its head the element $\diamondsuit_i \Psi_i$ of Φ^N and, immediately below, the respective values of N_i, C_i, A_i, and i. Table 2 is a parallel display of the generated terms of the sequence $\{K_i\}$ (which we name the *C-sequence*) and the partial productions Π_i.

7. Optimization (General)

The first stage of optimization of $\Pi(\Phi)$ consists in the elimination of redundant parentheses arising out of the transformations $T1$–$T8$. Having achieved the desired stratification of Φ, they shall now disappear from the scene, and in the following manner.

The production $\Pi(\Phi)$ is scanned, "back-to-front", one triple at a time. If and only if a triple $(C, \diamondsuit, \Psi)$ belongs to a segment of length 1 and $\diamondsuit \neq -$ is this triple eliminated from Π, and Ψ replaces the third member of its immediate predecessor. This "telescoping" procedure is based on the fact that a segment is of length 1 only if its "name" (i.e., current index C) is "addressed" by its immediate predecessor. This last assertion follows, in turn, from the first partial production rule.

TABLE 2

C-Sequence	Partial Productions
$K_1 = \Lambda$	$\Pi_1 = \Lambda$
$K_2 = (0)$	$\Pi_2 = \Pi_1(0, -, 1)$
$K_3 = (0, 1)$	$\Pi_3 = \Pi_2(1, *, 2)$
$K_4 = (0, 1, 2)$	$\Pi_4 = \Pi_3(2, **, 3)$
$K_5 = (0, 1, 2)$	$\Pi_5 = \Pi_4(3, \oplus, \text{XYZF})$
$K_6 = (0, 1, 2, 3)$	$\Pi_6 = \Pi_5(3, \oplus, 4)$
$K_7 = (0, 1, 2, 3, 4)$	$\Pi_7 = \Pi_6(4, +, 5)$
$K_8 = (0, 1, 2, 3, 4, 5)$	$\Pi_8 = \Pi_7(5, *, 6)$
$K_9 = (0, 1, 2, 3, 4, 5, 6)$	$\Pi_9 = \Pi_8(6, **, 7)$
$K_{10} = (0, 1, 2, 3, 4, 5, 6)$	$\Pi_{10} = \Pi_9(7, \oplus, \text{A})$
$K_{11} = (0, 1, 2, 3, 4, 5)$	$\Pi_{11} = \Pi_{10}$
$K_{12} = (0, 1, 2, 3, 4)$	$\Pi_{12} = \Pi_{11}$
$K_{13} = (0, 1, 2, 3)$	$\Pi_{13} = \Pi_{12}$
$K_{14} = (0, 1, 2)$	$\Pi_{14} = \Pi_{13}$
$K_{15} = (0, 1, 2, 3)$	$\Pi_{15} = \Pi_{14}(3, \oplus, 8)$
$K_{16} = (0, 1, 2, 3, 8)$	$\Pi_{16} = \Pi_{15}(8, +, 9)$
$K_{17} = (0, 1, 2, 3, 8, 9)$	$\Pi_{17} = \Pi_{16}(9, *, 10)$
$K_{18} = (0, 1, 2, 3, 8, 9, 10)$	$\Pi_{18} = \Pi_{17}(10, **, 11)$
$K_{19} = (0, 1, 2, 3, 8, 9, 10)$	$\Pi_{19} = \Pi_{18}(11, \oplus, \text{B})$
$K_{20} = (0, 1, 2, 3, 8, 9)$	$\Pi_{20} = \Pi_{19}$
$K_{21} = (0, 1, 2, 3, 8)$	$\Pi_{21} = \Pi_{20}$
$K_{22} = (0, 1, 2, 3, 8, 9)$	$\Pi_{22} = \Pi_{21}(9, *, 12)$
$K_{23} = (0, 1, 2, 3, 8, 9, 12)$	$\Pi_{23} = \Pi_{22}(12, **, 13)$
$K_{24} = (0, 1, 2, 3, 8, 9, 12)$	$\Pi_{24} = \Pi_{23}(13, \oplus, \text{C})$
$K_{25} = (0, 1, 2, 3, 8, 9)$	$\Pi_{25} = \Pi_{24}$
$K_{26} = (0, 1, 2, 3, 8, 9, 12)$	$\Pi_{26} = \Pi_{25}(12, **, 14)$
$K_{27} = (0, 1, 2, 3, 8, 9, 12, 14)$	$\Pi_{27} = \Pi_{26}(14, \oplus, 15)$
$K_{28} = (0, 1, 2, 3, 8, 9, 12, 14, 15)$	$\Pi_{28} = \Pi_{27}(15, -, 16)$
$K_{29} = (0, 1, 2, 3, 8, 9, 12, 14, 15, 16)$	$\Pi_{29} = \Pi_{28}(16, *, 17)$
$K_{30} = (0, 1, 2, 3, 8, 9, 12, 14, 15, 16, 17)$	$\Pi_{30} = \Pi_{29}(17, **, 18)$
$K_{31} = K_{30}$	$\Pi_{31} = \Pi_{30}(18, \oplus, \text{D})$
$K_{32} = (0, 1, 2, 3, 8, 9, 12, 14, 15, 16)$	$\Pi_{32} = \Pi_{31}$
$K_{33} = (0, 1, 2, 3, 8, 9, 12, 14, 15)$	$\Pi_{33} = \Pi_{32}$
$K_{34} = (0, 1, 2, 3, 8, 9, 12, 14)$	$\Pi_{34} = \Pi_{33}$
$K_{35} = (0, 1, 2, 3, 8, 9, 12)$	$\Pi_{35} = \Pi_{34}$
$K_{36} = (0, 1, 2, 3, 8, 9)$	$\Pi_{36} = \Pi_{35}$
$K_{37} = (0, 1, 2, 3, 8)$	$\Pi_{37} = \Pi_{36}$
$K_{38} = (0, 1, 2, 3)$	$\Pi_{38} = \Pi_{37}$
$K_{39} = (0, 1, 2)$	$\Pi_{39} = \Pi_{38}$
$K_{40} = (0, 1)$	$\Pi_{40} = \Pi_{39}$
$K_{41} = (0)$	$\Pi_{41} = \Pi_{40}$
$K_{42} = (0, 1)$	$\Pi_{42} = \Pi_{41}(1, /, 19)$
$K_{43} = (0, 1, 19)$	$\Pi_{43} = \Pi_{42}(19, **, 20)$
$K_{44} = K_{43}$	$\Pi_{44} = \Pi_{43}(20, \oplus, \text{E})$
$K_{45} = (0, 1)$	$\Pi_{45} = \Pi_{44}$
$K_{46} = (0)$	$\Pi_{46} = \Pi_{45}$
$K_{47} = \Lambda$	$\Pi_{47} = \Pi_{46}$
$K_{48} = (0)$	$\Pi_{48} = \Pi_{47}(0, -, 21)$
$K_{49} = (0, 21)$	$\Pi_{49} = \Pi_{48}(21, *, 22)$
$K_{50} = (0, 21, 22)$	$\Pi_{50} = \Pi_{49}(22, **, 23)$
$K_{51} = K_{50}$	$\Pi_{51} = \Pi_{50}(23, \oplus, \text{F})$
$K_{52} = (0, 21)$	$\Pi_{52} = \Pi_{51}$
$K_{53} = (0)$	$\Pi_{53} = \Pi_{52}$
$K_{54} = \Lambda$	$\Pi_{54} = \Pi_{53}$

Next, the set of so-condensed segments $\bar{\Pi}(\Phi)$ is ordered according to current indices, so that, if

$$\bar{\Pi}(\Phi) = S_1 \cdots S_L$$

where

$$S_c = (C, \Diamond_c^1, \Psi_c^1) \cdots (C, \Diamond_c^{\lambda_c}, \Psi_c^{\lambda_c})$$

then $S_{c'}$ "precedes" $S_{c''}$ if and only if $c' \leqq c''$.

The next stage of optimization involves the "elimination" of common subexpressions, so as to avoid redundant computation. This is accomplished in two steps:

1) Beginning with S_L, the last segment in $\bar{\Pi}(\Phi)$, and for each $i \leqq L$, the set of all S_j with $j < i$ is examined for the occurrence of an $S_j = S_i$. As soon as some $S_j = S_i$, S_j is eliminated from $\bar{\Pi}(\Phi)$, and all references to S_j replaced by references to S_i, i.e., if some $\Psi_k = j$, then j is set equal to i.

2) Having eliminated, by 1), common segments, we now eliminate common subexpressions. Beginning with S_L, and for each $i \leqq L$, the set of all S_j with $j < i$ is examined for the occurrence of more than one reference to S_i, i.e., the occurrence of Ψ_m, Ψ_n, with $m \neq n$ and $\Psi_m = \Psi_n = i$. If and only if this is the case is S_i tagged as a common subexpression (what we call a CS-*type* segment).

Procedures 1) and 2) together assure the elimination of outermost common subexpressions. Thus, if

$$\Phi = \text{A} * (\text{B} * \text{C}) + \text{SINF}(\text{A} * (\text{B} * \text{C})),$$

then

$$[\bar{\Pi}(\Phi) = (0, +, 1)(0, +, 14)(1, *, \text{A})(1, *, 7)$$
$$(7, *, \text{B})(7, *, \text{C})(14, \oplus, \text{SINF})(14, \oplus, 16)(16, *, \text{A})$$
$$(16, *, 22)(22, *, \text{B})(22, *, \text{C})]$$

Procedures 1) and 2) reduce $\bar{\Pi}$ to

$$(0, +, 16)(0, +, 14)(14, \oplus, \text{SINF})(14, \oplus, 16)$$
$$(16, *, \text{A})(16, *, 22)(22, *, \text{B})(22, *, \text{C}),$$

with S_{16} tagged as a CS-type segment, since $\Psi_0^1 = \Psi_{14}^2 = 16$.

We shall denote the result of common subexpression elimination by $(\bar{\bar{\Pi}}\Phi)$.

8. Optimization (Special)

Owing to the fact that the FORTRAN System was originally designed to compile "object" (running) programs in 704 language, certain further species of optimization regarding the compilation of arithmetic statements appear to be necessary if advantage is to be taken of the machine's own special characteristics. We list these in the order in which they are considered by the executive program.

1) Each segment S_i with $\Diamond_i^1 = *$ is scanned for possible permutation of its members, so as to minimize the occurrence of compiled memory accesses. Specifically, each segment S_i of the form

$$(i, *, \Psi_i^1) \cdots (i, \Diamond_i^{\lambda_i}, \Psi_i^{\lambda_i})$$

where

$$\Diamond_i^j = * \text{ or } /, \qquad 1 < j \leqq \lambda_i$$

undergoes permutation of its elements so as to yield a (possibly null) maximal subsegment of the form

$$(i, *, \Psi_i^{j_1})(i, /, \Psi_i^{j_2}) \cdots (i, *, \Psi_i^{j_{k-1}})(i, /, \Psi_i^{j_k})$$

i.e., a maximal subsegment whose operator structure is

$$*, /, */, \cdots, */.$$

Since the only remaining elements (if any) are of the form $(i, *, \Psi)$ or $(i, /, \Psi)$, consider the following cases:

(*i*) The number of $*$'s in S_i is one more than the number of /'s. In this case, the operator structure of S_i is $*/*/ \cdots */*$.

(*ii*) The number of $*$'s in S_i is at least two more than the number of /'s. In this case, the operator structure of S_i is $**/*/ \cdots */* \cdots *$.

(*iii*) The number of /'s in S_i exceeds the number of $*$'s. In this case, the structure is $*/ \cdots */ \cdots /$.

2) A segment S_i is said to be *type* MQ if its last operation is /; otherwise, it is said to be *type* AC.

A further species of optimization of $\bar{\bar{\Pi}}(\Phi)$, which we term *linkage*, designed to minimize memory accesses, is performed in the following manner. Beginning with the last segment, S_L, each segment S_i is examined as to type and affected in the following ways.

(I) S_i is type AC. Then S_i is tagged as AC-*linkable* and S_{i-1} as AC-*linked*, if and only if one of the following conditions obtains:

(i) $\Diamond_{i-1}^{1} = +$ or $-$, and for some j, $\Psi_{i-1}^{j} = i$. In this case, in addition to tagging S_i and S_{i-1}, interchange the first and jth elements of S_{i-1}.

(ii) $\Diamond_{i-1}^{1} = *$, $\Diamond_{i-1}^{2} = /$, and for some j, $\Psi_{i-1}^{j} = i$, $\Diamond_{i-1}^{j} = *$. Again, in addition to tagging S_i and S_{i-1}, interchange the first and jth elements of S_{i-1}.

(iii) $\Diamond_{i-1}^{1} = \oplus$, $\Psi_{i-1}^{2} = i$ and Ψ_{i-1}^{1} is the name of a *closed subroutine* (see below), and FN-type function, or an open univariate function.

(iv) $\Diamond_{i-1}^{1} = **$ and $\Psi_{i-1}^{1} = i$.

(II) S_i is type MQ. Then S_i is tagged as MQ-*linkable* and S_{i-1} as MQ-*linked*, if and only if one of the following conditions obtains:

(i) $\Diamond_{i-1}^{1} = *$, $\Diamond_{i-1}^{2} = *$ and for some j, $\Psi_{i-1}^{j} = i$, $\Diamond_{i-1}^{j} = *$. In this case, in addition to tagging S_i and S_{i-1}, interchange the first and jth elements of S_{i-1}.

(ii) $\Diamond_{i-1}^{1} = \oplus$, $\Psi_{i-1}^{3} = i$ and Ψ_{i-1}^{1} is the name of a closed subroutine (see below), an FN-type function or an open univariate function.

(iii) $\Diamond_{i-1}^{1} = **$. There are two cases:

(a) Ψ_{i-1}^{2} is the name of an integer constant less than 7 (in which case S_{i-1} is compiled as an open subroutine), and $\Psi_{i-1}^{1} = i$.

(b) $\Psi_{i-1}^{2} = i$.

In all other cases, i.e. cases which do not fall either under (I) or (II) above, S_i is *unlinkable* and S_{i-1} *unlinked*.

9. Function Types

With each library or FN-type function appearing in a FORTRAN program is associated a type number according to the following scheme:

I) Each library function is of type 0.

II) If Φ is an FN-function name, where

$$\Phi(A_1, \cdots, A_n) = E[A_1, \cdots, A_n] \dashv,$$

then

(a) if E contains no library or FN-function name, Φ is of type 1.

(b) if E contains a library or FN-function name, and $\tau_1, \cdots, \tau_k$ are the type numbers already associated with these functions, then the type number of Φ is simply

$$\max(\tau_1, \cdots, \tau_k) + 1.$$

10. Address Compilation

Each member of an element (triple) occurring in $\bar{\bar{\Pi}}(\Phi)$ is represented during compilation by the contents of a full word of 704 storage. These three words are referred to, respectively, as the *tag word*, *operator word*, and *symbol word*. The precise bit-structure of each of these words is a function of the role played by the element in question.

The tag word of each element of a segment S_i contains not only the current index i, but in addition, if this element refers to a subscripted variable in the original (unreduced) expression Φ a set of three tags identifying the dimension, subscript and addend combinations.

The operator word of each element contains the operation code and, if the first element of a segment S_i, a set of bits containing information as to certain properties of this segment, viz., whether the segment is linkable or linked (and through which arithmetic register); whether arithmetic for this segment is floating or integer mode; whether, if $\Diamond_i^{1} = \oplus$, S_i defines a library, open subroutine, FS- or FN-function (and, if the latter, what its type); whether or not S_i defines a common subexpression; whether the result of computation of S_i appears in the accumulator or multiplier-quotient register.

The symbol word of each element contains the name of the operand, which may be an integer or floating mode variable, an integer or floating mode constant, a function, or some other segment S_j.

We shall denote the compiled tagged-address associated with the jth symbol of the ith segment by $\bar{\Psi}_i^{j}$.

Actual SAP-form address compilation proceeds as follows:

(i) An address reference to an integer (respectively floating-point) constant is compiled into a symbolic address 2) (respectively 3)) and relative address ν (respectively μ), where ν (respectively μ) is associated with the νth (respectively μth) distinct integer (respectively floating-point) constant occurring in a given source program.

(ii) An address reference to a subscripted variable is compiled as follows:

(a) If $K * \Sigma \pm P$ is the canonical form of the subscript associated with a one-dimensional variable Ψ_i^{j}, then the symbolic address is compiled as Ψ_i^{j} and the relative address as $1 \mp P$.

(b) If $K_1 * \Sigma_1 \pm P_1$, $K_2 * \Sigma_2 \pm P_2$ are the canonical subscripts associated with a two-dimensional variable Ψ_i^{j}, then the symbolic address is compiled as Ψ_i^{j} and the relative address as

$$\epsilon - (\pm P_1 - \epsilon_1) - \Gamma_1(\pm P_2 - \epsilon_2),$$

where

$$\epsilon = \begin{cases} 0 & \text{if } \Sigma_1 = \Sigma_2 = \Lambda \\ 1 & \text{otherwise,} \end{cases}$$

$$\epsilon_k = \begin{cases} 0 & \text{if } \Sigma_k \neq \Lambda \\ 1 & \text{otherwise,} \end{cases}$$

and Γ_1 is the first dimension of Ψ_i^{j}.

(c) Finally, if $K_1 * \Sigma_1 \pm P_1$, $K_2 * \Sigma_2 \pm P_2$,

$$K_3 * \Sigma_3 \pm P_3$$

are the canonical subscripts associated with the three-dimensional variable Ψ_i^j, then the symbolic address is compiled as Ψ_i^j and the relative address as

$$\epsilon - (\pm P_1 - \epsilon_1) - \Gamma_1(\pm P_2 - \epsilon_2) - \Gamma_1\Gamma_2(\pm P_3 - \epsilon_3)$$

where

$$\epsilon = \begin{cases} 0 & \text{if } \Sigma_1 = \Sigma_2 = \Sigma_3 = \Lambda \\ 1 & \text{otherwise,} \end{cases}$$

$$\epsilon_k = \begin{cases} 0 & \text{if } \Sigma_k \neq \Lambda \\ 1 & \text{otherwise,} \end{cases}$$

and Γ_1, Γ_2 are the first two dimensions of Ψ_i^j.

(iii) An address reference to another segment S_j is compiled into a symbolic address 1) τ, where τ is the type number associated with the arithmetic expression $= \Phi \dashv$, and a relative address φ_i. The φ_i's are *erasable* storage relative addresses determined in the following way: Beginning with the last segment S_L of $\overline{\overline{\Pi}}(\Phi)$, each S_i is examined to determine whether it is nonlinkable or is tagged as a common subexpression. In either case (and only then) an erasable storage relative address

$$\varphi_i = \varphi_j + 1$$

is associated with S_i, where S_j $(i \leqq j)$ is the last segment nonlinkable or tagged as a common subexpression, and, initially, $\varphi = -1$ or 0, depending upon whether $\overline{\overline{\Pi}}(\Phi)$ does or does not define an FN-type function.

(iv) With each tape library or FN-type function is associated a class of erasable storage cells set aside as a buffer for the transmission of its arguments. The type τ of any given class is determined by the type assigned to the function in question. Thus, tape library functions are always of type 0, and an FN-function is of type 1 greater than the highest type occurring in its definition.

In the case of a tape library function, an address reference to its kth argument is compiled into a symbolic address 4)0 and a relative address $-(k-1)$.

In the case of an FN-function, on the other hand, an address reference to its kth argument is compiled into a symbolic address 4)τ, where τ is the type number, and relative address k-1.

It should be noted, at this point, that the necessity for typing tape library and FN-functions arises from the fact that either may occur within the definition of an FN-function. Unrestricted nesting of these functions within such a context is possible, therefore, only if their argument buffer regions are non-overlapping.

(v) For the reasons cited at the end of (iv), the class of calling-index saving cells is also typed, type 0 assigned for tape library functions and type $\tau > 0$ for a given FN-function of type τ. The relative address in both cases is 0, the symbolic address 7)0 for tape library functions and 7)τ for FN-functions.

(vi) Finally, when intrasegment erasable storage is required, a single cell is set aside having symbolic address 7)0 and relative address 0.

11. Arithmetic Statement Compilation

Beginning with the last segment S_L of $\overline{\overline{\Pi}}(\Phi)$, each segment S_i is "forward scanned" and compiled according to the following schema.

(I) *Initial Compilation of a Segment*

(A) $\Diamond_i^1 = +$. There are two cases:

(i) S_i is linked. Proceed to (II), unless S_i is of length 1, in which case proceed to (III).

(ii) S_i is unlinked. Compile CLA $\bar{\Psi}_i^1$ and proceed to (II).

(B) $\Diamond_i^1 = -$. Again, two cases:

(i) S_i is linked. Compile CHS and proceed to (II), unless S_i is of length 1, in which case proceed to (III).

(ii) S_i is unlinked. Compile CLS $\bar{\Psi}_i^1$, then proceed as per B(i).

(C) $\Diamond_i^1 = *$. Two cases:

(i) S_i is linked. Proceed to (II).

(ii) S_i is unlinked. Two subcases:

(a) $\Diamond_i^2 = /$. Compile CLA $\bar{\Psi}_i^1$.

(b) $\Diamond_i^2 = *$. Compile LDQ $\bar{\Psi}_i^1$.

In either case, proceed next to (II).

(D) $\Diamond_i^1 = \oplus$. There are several cases:

(i) Ψ_i^1 is the name of a tape library subroutine. Three subcases:

(a) S_i is AC-linked. Compile the sequence

$$\left\{\begin{array}{l} \left\{\begin{array}{l} \text{LDQ } \bar{\Psi}_i^4 \\ \text{STQ } 4)0 - 2 \\ \cdots\cdots\cdots \\ \text{LDQ } \bar{\Psi}_i^{\lambda_i} \\ \text{STQ } 4)0 - (\lambda_i - 2) \end{array}\right. \\ \text{LDQ } \bar{\Psi}_i^3 \end{array}\right.$$

followed by the sequence

SXD 7)0,4

TSX Ψ_i^1,4

LXD 7)0,4.

Either subsequence in braces is vacuous (not compiled) in the event S_i defines a univariate or bivariate function only. Note, further, that both the SXD and LXD instructions surrounding the TSX may be eliminated by a later section of the FORTRAN executive system in the event that the flow of indexing information obviates saving the contents of register 4 at this point.

(b) S_i is MQ-linked. Compile the sequence

$$\left\{\begin{array}{l} \text{CLA } \Psi_i^4 \\ \text{STO } 4)0 - 2 \\ \cdots\cdots\cdots \\ \text{CLA } \bar{\Psi}_i^{\lambda_i} \\ \text{STO } 4)0 - (\lambda_i - 2) \end{array}\right.$$

followed by the sequence

CLA $\bar{\Psi}_i^2$

SXD 7)0,4

TSX Ψ_i^1,4

LXD 7)0,4.

The subsequence in braces is vacuous in the event S_i defines a bivariate function only.

(c) S_i is unlinked. Compile CLA $\bar{\Psi}_i^2$, then the sequence

$$\left\{\begin{array}{l}\left\{\begin{array}{ll}\text{LDQ} & \bar{\Psi}_i^4\\ \text{STQ} & 4)0-2\\ \cdots\cdots & \\ \text{LDQ} & \bar{\Psi}_i^{\lambda_i}\\ \text{STQ} & 4)0-(\lambda_i-2)\end{array}\right.\\ \begin{array}{ll}\text{LDQ} & \bar{\Psi}_i^3\end{array}\end{array}\right.$$

followed by the sequence

$$\begin{array}{l}\text{SXD } 7)0,4\\ \text{TSX } \Psi_i^1,4\\ \text{LXD } 7)0,4.\end{array}$$

Either subsequence in braces is vacuous in the event S_i defines a univariate or bivariate function only.

(ii) Ψ_i^1 is the name of an FN-function of type τ. Three subcases:

(a) S_i is AC-linked. Compile STO $4)\tau$, then the sequence

$$\left\{\begin{array}{l}\begin{array}{ll}\text{LDQ} & \bar{\Psi}_i^3\\ \text{STQ} & 4)\tau+1\end{array}\\ \left\{\begin{array}{ll}\text{CLA} & \bar{\Psi}_i^4\\ \text{STO} & 4)\tau+2\\ \cdots\cdots & \\ \text{CLA} & \bar{\Psi}_i^{\lambda_i}\\ \text{STO} & 4)\tau+(\lambda_i-2)\end{array}\right.\end{array}\right.$$

followed by the sequence

$$\begin{array}{l}\text{SXD } 7)\tau,4\\ \text{TSX } \Psi_i^1,4\\ \text{LXD } 7)\tau,4.\end{array}$$

Either subsequence in braces is vacuous in the event S_i defines a univariate or bivariate function only.

(b) S_i is MQ-linked. Compile the sequence

$$\begin{array}{l}\text{CLA } \bar{\Psi}_i^2\\ \text{STO } 4)\tau\\ \text{STQ } 4)\tau+1\end{array}$$

then the sequence

$$\begin{array}{ll}\text{CLA} & \bar{\Psi}_i^4\\ \text{STO} & 4)\tau+2\\ \cdots\cdots & \\ \text{CLA} & \bar{\Psi}_i^{\lambda_i}\\ \text{STO} & 4)\tau+(\lambda_i-2)\end{array}$$

followed by the sequence

$$\begin{array}{l}\text{SXD } 7)\tau,4\\ \text{TSX } \Psi_i^1,4\\ \text{LXD } 7)\tau,4.\end{array}$$

The subsequence in braces is vacuous in the event S_i defines a bivariate function only.

(c) S_i is unlinked. Compile the sequence

$$\begin{array}{l}\text{CLA } \bar{\Psi}_i^2\\ \text{STO } 4)\tau,\end{array}$$

then the sequence

$$\left\{\begin{array}{l}\begin{array}{ll}\text{LDQ} & \bar{\Psi}_i^3\\ \text{STQ} & 4)\tau+1\end{array}\\ \left\{\begin{array}{ll}\text{CLA} & \bar{\Psi}_i^4\\ \text{STO} & 4)\tau+2\\ \cdots\cdots & \\ \text{CLA} & \bar{\Psi}_i^{\lambda_i}\\ \text{STO} & 4)\tau+(\lambda_i-2)\end{array}\right.\end{array}\right.$$

followed, again, by the sequence

$$\begin{array}{l}\text{SXD } 7)\tau,4\\ \text{TSX } \Psi_i^1,4\\ \text{LXD } 7)\tau,4.\end{array}$$

Either subsequence in braces is vacuous in the event S_i defines a univariate or bivariate function only.

(iii) Ψ_i^1 is the name of an FS-function. Two subcases:

(a) No subscripted variable containing a variable subscript index occurs as an argument of Ψ_i^1. Compile the sequence

$$\begin{array}{l}\text{SXD } 7)0,4\\ \text{TSX } \Psi_i^1,4\\ \text{———}\bar{\Psi}_i^2\\ \cdots\\ \text{———}\bar{\Psi}_i^{\lambda_i}\\ \text{LXD } 7)0,4.\end{array}$$

(b) Some subscripted variable occurring as an argument of Ψ_i^1 contains a variable subscript index. If $\Psi_i^{j_1}, \cdots, \Psi_i^{j_k}$ comprise such a set, then compile the sequence

$$\begin{array}{l}\text{PXD } \bar{\Psi}_i^{j}\\ \text{ARS } 18\\ \text{ADD } *-2\\ \text{STA } \alpha+j_1\\ \cdots\\ \text{PXD } \bar{\Psi}_i^{j_k}\\ \text{ARS } 18\\ \text{ADD } *-2\\ \text{STA } \alpha+j_k\\ \text{SXD } 7)0,4\\ \text{TSX } \Psi_i^1,4\\ \cdots\\ \text{LXD } 7)0,4.\end{array}$$

The symbolic address α denotes the relative location of the TSX instruction within the body of the program, and each entry in the sequence between the TSX and LXD instructions is of the form $\bar{\Psi}_i^m$, if $m \neq$ any j_n, or Ψ^{j_m} if otherwise. We recall, in passing, that in this context $\bar{\Psi}_i^{j_n}$ is our symbol for the composite symbolic-relative address and tag of Ψ^{j_n} if the latter denotes a subscripted variable. That is, if $\rho_i^{j_n}$ is the algebraic (signed) relative address and $\tau_i^{j_n}$ the tag associated with this variable, then

$$\bar{\Psi}^{j_n} = \Psi_i^{j_n} + \rho_i^{j_n}, \tau_i^{j_n}.$$

Thus, the effect of the sequence PXD, ARS, ADD, STA is to compute the effective address of this variable and store same as an actual address in the calling sequence, which address is then available to the subroutine defining the value of Ψ_i^1 via the calling index register 4. The symbolic address * denotes the contents of the 704 program counter at the time an instruction having this symbolic address is interpreted by the machine.

(iv) Ψ_i^1 is the name of a built-in open subroutine. Owing to the fact that compilation of an open subroutine into the main body of an object program is an essentially ad hoc procedure—depending, as it does, on the particular nature of the function in question, the number of its arguments, and upon the particular context within which the function arises—and since, further, actual compilation of open subroutines is deferred to a section of the executive system later than that with which the present paper is concerned, we shall omit a detailed description of this subject for the present.

(E) $\Diamond_i^1 = **$.

(i) S_i defines an open subroutine, viz., Ψ_i^2 is the name of an integer constant less than 7. The same remarks apply here as for (D) (iv) (q.v.).

(ii) S_i defines a closed subroutine.

(a) S_i is AC-linked. Compile LDQ $\bar{\Psi}_i^2$; then proceed to (C).

(b) S_i is not AC-linked. Compile CLA $\bar{\Psi}_i^1$.

1) S_i is MQ-linked. Proceed to (C).

2) S_i is not MQ-linked. Proceed to (a).

(c) Compile SXD 7)0,4; then proceed to (d).

(d) Three distinct built-in tape library subroutines compute the exponential according as the exponent is integer or floating valued, or the base is fixed valued.

(1) Ψ_i^1, Ψ_i^2 are both integer valued. Compile TSX EXP(1,4.

(2) Ψ_i^1 is floating valued, Ψ_i^2 integer valued. Compile TSX EXP(2,4.

(3) Ψ_i^1, Ψ_i^2 are both floating valued. Compile TSX EXP(3,4.

(4) Ψ_i^1 integer valued, Ψ_i^2 floating valued. Disallowed.

(e) Finally, compile LXD 7)0,4 and proceed to (III).

(II) *Intra-Segment Compilation*

(A) $\Diamond_i^j = +$. Two cases:

(i) S_i is in floating-point mode. Compile FAD $\bar{\Psi}_i^j$.

(ii) S_i is in integer mode. Compile ADD $\bar{\Psi}_i^j$.

(B) $\Diamond_i^j = -$. Two cases:

(i) S_i is in floating-point mode. Compile FSB $\bar{\Psi}_i^j$.

(ii) S_i is in integer mode. Compile SUB $\bar{\Psi}_i^j$.

(C) $\Diamond_i^j = *$. Two cases:

(i) Predecessor in MQ. I.e., $\Diamond_i^{j-1} = /$ or $\Diamond_i^1 = *$ and $j = 2$. Two subcases:

a) S_i floating-point. Compile FMP $\bar{\Psi}_i^j$

b) S_i integer. Compile MPY $\bar{\Psi}_i^j$, ALS 17.

(ii) Predecessor in ACC. I.e., $\Diamond_i^{j-1} = *$ and $j \neq 2$ Compile STO 7)0, LDQ 7)0, and proceed to (i) (a) or (i) (b) above, depending upon mode of S_i.

(D) $\Diamond_i^j = /$. Two cases:

(i) Predecessor in MQ. I.E. $\Diamond_i^{j-1} = /$. Two subcases

(a) S_i floating-point. Compile STQ 7)0, CLA 7)0, FDP $\bar{\Psi}_i^j$.

(b) S_i integer. Compile DVP $\bar{\Psi}_i^j$, CLM, LLS 18.

(ii) Predecessor in ACC. I.e. $\Diamond_i^{j-1} = *$. Two subcases

(a) S_i floating-point. Compile FDP $\bar{\Psi}_i^j$.

(b) S_i integer. Compile LRS 35 and proceed to (i) (b) above.

(III) *Final Compilation of a Segment*

(A) Last segment compiled was S_0.

(i) $\bar{\bar{\Pi}}(\Phi)$ is an IF-type production, i.e., is the production of an expression Φ contained in a FORTRAN source language statement of the form

$$\text{IF}(\Phi)n_1, n_2, n_3$$

where the n_i's are source program statement names.

(a) S_0 is type AC. Finis.

(b) S_0 is type MQ. Compile LLS 37 and finis.

(ii) $\bar{\bar{\Pi}}(\Phi)$ is a CALL-type production, i.e., is the production of an expression Φ contained in a FORTRAN source language stagement of the form CALL Φ, where Φ is an FS-function. Finis.

(iii) $\bar{\bar{\Pi}}$ is neither an IF- nor CALL-type production Then the source language statement containing Φ is of the form $\Psi = \Phi \dashv$, where Ψ is a variable or FN-function name. Consider the cases:

(a) Ψ is integer-valued.

(1) S_0 is in floating mode and is type AC. Compile the (fixing) sequence UFA 6), LRS 0, ANA 6) + 1, LLS 0, ALS 18 We note, here, that two constants having SAP identifications 6), 6)+1, $\cdots$ are compiled into the object program constant-region. These constants are, in 704 octal word-format, 233000000000 and 000000077777, respectively. Thus, the above sequence has the effect of placing the point of the floating-point number, whose integer form is desired, to the extreme right of the accumulator preserving its sign in the MQ register, extracting the mantissa (now positioned in the last 15 bits of the accumulator), restoring the sign and shifting the mantissa into the decrement field.

(α) If Ψ is an FN-function name, compile TRA 1,4 Finis.

(β) If not, compile STO $\bar{\Psi}$, and finis.

(2) S_0 is in floating mode and is type MQ. Compile STQ 7)0, CLA 7)0, then proceed as in (a) (1), above.

(3) S_0 is integer valued.

(α) Ψ is an FN-function name.

a) S_0 is type AC. Compile TRA 1,4.

b) S_0 is type MQ. Compile STQ 7)0, CLA 7)0, TRA 1,4.

(β) Ψ is a variable.

a) S_0 is type AC. Compile STO $\bar{\Psi}$.

b) S_0 is type MQ. Compile STQ $\bar{\Psi}$.

(b) Ψ is floating valued.

(1) S_0 is floating valued. Proceed as in (a) (3), above.

(2) S_0 is integer valued.

(α) S_0 is type AC. Compile LRS 18, ORA 6), FAD 6).

a) Ψ is an FN-function name. Compile TRA 1,4.

b) Otherwise, compile STO $\bar{\Psi}$.

(β) S_0 is type MQ. Compile STQ 7)0, CLA 7)0, and proceed as in (2) (α) above.

B) Last segment (S_i) compiled was not S_0.

(i) S_i linkable and not a common subexpression. Proceed to compilation of S_{i-1}.

(ii) S_i not linkable or is a common subexpression.

(a) S_i is type AC. Compile STO 1)τ + φ_i, where τ is the type number associated with Φ, and equals 0 if Ψ is not an FS-function name; otherwise, τ is the type of the function currently being defined (see section 9). The relative address φ_i is defined as in section 10 (iii). Proceed to compilation of S_{i-1}.

(b) S_i is type MQ. Compile STQ 1)τ + φ_i, and proceed to compilation of S_{i-1}.

APPENDIX

A1. Implicit Multiplication

A certain conciseness of notation in the writing of expressions is allowed of by the fact than an $*$ sign need not occur in an expression Φ if Φ is not of the form $\Psi * X$, where $T_1(\Psi)$, $H_1(X)$ belong to $\mathfrak{B}_\Phi$. Thus, if $\Phi \sim \Psi$ (read "Φ equivalent to Ψ") is taken to mean that the corresponding arithmetic expressions $= \Phi \dashv$, $= \Psi \dashv$ yield identical *object* (machine-language) programs, then

$$(-\text{A})\text{B}(\text{I}) \sim (-\text{A})*\text{B}(\text{I})$$

$$\text{X01(I)B} \sim \text{X01(I)}*\text{B}$$

$$\text{SINF(X)COSF(X)} \sim \text{SINF(X)}*\text{COSF(X)}$$

$$\text{A(B + C)} \sim \text{A}*(\text{B + C})$$

$$(\text{A/B})\text{LOGF(X)} \sim (\text{A/B})*\text{LOGF(X)}$$

$$(\text{A + B})(\text{X + Y}) \sim (\text{A + B})*(\text{X + Y})$$

$$\text{TANF(X)}(\text{A} - \text{B}) \sim \text{TANF(X)}*(\text{A} - \text{B})$$

A2. Boolean Statements

An immediate extension of the mechanisms of arithmetic statement compilation, exploiting the AND-OR logic of the 704, is readily at hand. If the operation signs $+$, $*$, and $-$ are interpreted to denote union, intersection and complementation, respectively, then a certain subset of the set of expressions defined by *E1–E6* (see section 3) is sufficient for the formulation of any Boolean function on the set of all 36-bit binary strings. We shall call the elements of this subset *sentences* and recursively characterize them as follows:

*S*1. Every floating-point variable name Φ is a sentence. If, furthermore, $\Phi = \varphi_1, \cdots \varphi_n$, and $L(\Phi) < 4$ or $\varphi_n \neq$ F, and Φ appears in a DIMENSION statement, and if $\Sigma_1, \cdots, \Sigma_k$ ($1 \leqq k \leqq 3$) are subscripts, then $\Phi(\Sigma_1, \cdots, \Sigma_k)$ is a sentence.

*S*2. If Φ is a sentence, so is (Φ).

*S*3. If Φ is a sentence such that $H_1(\Phi) \neq -$, and Φ is not of the form $\Psi + X$, where Ψ, X are sentences, then $-\Phi$ is a sentence.

*S*4. If Φ is a sentence of the form $\Psi + X$, where Ψ, X are sentences, then $-(\Phi)$ is a sentence.

*S*5. If Φ is an n-adic function name with $H_1(\Phi) \neq$ X, and $A_1, \cdots, A_n$ are sentences, then $\Phi(A_1, \cdots, A_n)$ is a sentence.

*S*6. If Φ, Ψ are sentences, and $H_1(\Phi) \neq -$, $H_1(\Psi) \neq -$, then $\Phi + \Psi$, $\Phi * \Psi$ are sentences.

(Note: The same rule regarding implicit multiplication of expressions—mentioned in *A1* above—applies as well to the construction of sentences.)

Rules *S1–S6* prohibit, by implication, the writing of expressions of the form $\Phi - \Psi$, where Φ, Ψ are sentences. Thus, what in conventional logical notation is written $\Phi \cap \Psi'$ or $\Phi\Lambda \sim \Psi$ cannot be abbreviated to $\Phi - \Psi$, but must be rendered by $\Phi * (-\Psi)$.

We define a *Boolean expression* as a string of the form

$$= \Phi \dashv$$

where Φ is a sentence.

Similarly, a *pure Boolean statement* is a string of the form

$$\Psi = \Phi \dashv$$

where Ψ is a subscripted or nonsubscripted floating-point variable, and $= \Phi \dashv$ is a Boolean expression.

A *quasi-Boolean statement* is a string of one of the following forms

(a) IF (Φ), where Φ is a sentence.

(b) CALL $\Phi(A_1, \cdots, A_n)$, where Φ is a function name such that

(i) if $\Phi = \varphi_1 \cdots \varphi_n$, then $L(\Phi) < 4$ or $\varphi_n \neq$ F, Φ does not appear in a DIMENSION statement, and $L(\Phi) \leqq 6$.

(ii) each A_i is a sentence or a Hollerith field.

A (*Boolean*) *function definition* is a string of the form

$$\Phi(A_1, \cdots, A_n) = E[A_1, \cdots, A_n]$$

where Φ is a function name, and such that

(i) if $\Phi = \varphi_1 \cdots \varphi_n$, then $H_1(\Phi) \neq$ X, $4 \leqq L(\Phi) \leqq 7$ and $\varphi_n =$ F.

(ii) each A_i is a nonsubscripted floating-point variable name.

(iii) $E[A_1, \cdots, A_n]$ is a sentence in the (free) variables $A_1, \cdots, A_n$, wherein each A_i may occur vacuously.

Exactly the same reduction, level analysis and general optimization procedures are applied to Boolean as to arithmetic expressions. In addition, the special optimization procedures (section 8) apply with one minor modification. Since each segment S_i is type AC (the / sign cannot occur in a sentence), then linkage can only occur, if ever, through the machine's accumulator register. Hence, if $\Diamond_{i-1}^{1} = *$, and for some j, $\Psi_{i-1}^{j} = i$, interchange the first and jth elements of S_{i-1}, tagging S_i as *AC-linkable* and S_{i-1} as *AC-linked*.

Compilation of Boolean statements proceeds in a manner analogous to that of arithmetic statements except for the following operation code transformations:

```
CLA → CAL
LDQ → CAL
STO → SLW
CLS → CAL, COM
FMP → ANA
FAD → ORA
CHS → COM
STQ → SLW
```

EXAMPLE.

```
IMPF(X, Y) = (−X) + Y
EQUIVF(X, Y) = IMPF(X, Y)IMPF(Y, X)
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Z = (((A*B)C) + D) + IMPF(((A*B)C) + D),
        EQUIVF(−(−X), Y))
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

We shall assume that the above statements appear in a possibly more extensive program, and that each is tagged as a Boolean-type statement (the letter B in column 1 of an IBM FORTRAN card).

Note that the IMPF function is type 1 and translates into the following instruction sequence:

```
CAL 4)1
COM
ORA 4)1 + 1
TRA 1, 4
```

The (free) variables X, Y are associated with 4)1, 4) 1 + 1, respectively, in this case.

The EQUIVF function is type 2, and translates into

```
CAL 4)2 + 1
SLW 4)1
CAL 4)2
SLW 4)1 + 1
SXD 7)1, 4
TSX IMP,4
LXD 7)1,4
SLW 1)2
CAL 4)2
SLW 4)1
CAL 4)2 + 1
SLW 4)1 + 1
SXD 7)1,4
TSX IMP,4
LXD 7)1,4
ANA 1)2
TRA 1,4
```

Finally, the third of the above statements translates into the sequence

```
CAL X
COM
COM
SLW 4)2
CAL Y
SLW 4)2 + 1
TSX EQUIV,4
SLW 1)1 + 1
CAL A
ANA B
ANA C
ORA D
SLW 1) + 2
SLW 4)1
CAL 1) + 1
SLW 4)1 + 1
TSX IMP,4
ORA 1) + 2
SLW Z
```