An Algorithm for Coding Efficient Arithmetic Operations

Robert W. Floyd

Armour Research Foundation of Illinois Institute of Technology, Chicago, Illinois

Abstract. Most existing formula translation schemes yield inefficient coding. A method is described which reduces the number of store and fetch operations, evaluates constant subexpressions during compilation, and recognizes many equivalent subexpressions.

Most previously published algorithms for formula translation depend upon a left-to-right scan of the formula, during which each symbol encountered either causes generation of coding or is saved on a list until it can be correctly interpreted in context. An exception is the GAT translator, which scans from right to left. In coding arithmetic operators for machines with accumulators (i.e., one- or two-address machines), right-to-left scans potentially generate more efficient coding than left-to-right scans. The reason can be seen by considering the formula

$$x := (u\ \theta_1\ v)/(y\ \theta_2\ z),$$

where the θ's are arbitrary arithmetic operators, each implemented by a single machine instruction. The symbolic coding for a typical one-address machine generated by both types of scan is shown below:

Left-to-right	*Right-to-left*
CLA u	CLA y
θ_1 v	θ_2 z
STR T_1	STR T_1
CLA y	CLA u
θ_2 z	θ_1 v
STR T_2	DIV T_1
CLA T_1	STR x
DIV T_2	
STR x	
(9 instructions)	(7 instructions)

Generally, it is desirable to compute first the right hand argument of a division or subtraction.

When formulae contain subscripted variables, however, a pure right-to-left scan is not yet the most efficient coding process, whether or not index registers are used. Assume a machine with an index register, IXR1. If θ_2 is a commutative operator such as + or ×, the formula

$$x := u\ [i\ \theta_1\ j]\ \theta_2\ (v\ \theta_3\ w)$$

may be coded in two ways:

Right-to-Left	*Best coding*
CLA v	CLA i
θ_3 w	θ_1 j
STR T_1	STR IXR1
CLA i	CLA v
θ_1 j	θ_3 w
STR IXR1	θ_2 U_0 , 1
CLA U_0 , 1	STR x
θ_2 T_1	
STR x	
(9 instructions)	(7 instructions)

The process by which efficient coding is written for formulae containing subscripted variables is a bi-directional scan. The statement is examined, character by character, from left to right. During this examination, identifiers and numerical constants are replaced by single symbols. When a right bracket is encountered, a right-to-left scan is initiated which codes the subscript or subscript list, terminating either by storing the result in an index register or by performing an address modification upon an instruction. The subscripted variable is then replaced in the formula by the symbol of the result of the coding just generated. When a statement terminator (; or **end** or **else** in ALGOL) is encountered, a right-to-left scan is initiated which completes the coding of the statement. The net effect may be loosely described by saying that subscripts are coded first in an otherwise right-to-left scan.

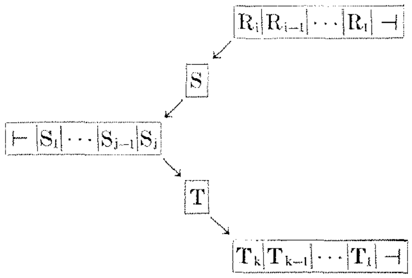
For many machines best results are obtained if the above process is used to obtain a list of two-address pseudocodes, and the machine or assembly language coding is then generated from the pseudocodes in reverse order, interpreting the last pseudocode first. Before any pseudocode is placed on the list, its operation and operand pair are compared to all previous pseudocodes of the current formula; if it has appeared before, it need not be rewritten on the list, and the symbol for its result may be replaced in the formula by the symbol for the result of the earlier code. At the time that machine instructions are generated from the pseudocodes, if the symbol for the ith partial result appears as an operand in the jth pseudocode, and i is less than $j-1$, the ith pseudocode must be marked in one bit position, to show that its result will be used at a later stage and must therefore be transmitted to temporary storage. Examples 1 and 2 contain examples of pseudocode lists.

A subscript list may consist of an arbitrarily large number of subscripts. Since each pseudocode may contain at most two operands, it is necessary to allow an arbitrarily large number of pseudocodes for each subscripting operation. The convention adopted here represents x[i,j,k] by the pseudocodes

```
,  k  0
,  j  0
,  i  0
[  x  0
```

Each pseudocode whose operator is a comma is associated with the next pseudocode whose operator is a left bracket. Subscript names appear as first operands with comma operators; array names, with left bracket operators. Functions of two or more variables are not considered here, but could be treated in the same way.

Storage of the symbols of the formula being encoded will require three push-down or "yo-yo" lists, and two fixed locations. The structure of these lists, and the flow of information between them, may be diagrammed as follows:


R_i R_{i-1} ··· R_1 ⊣
S
⊢ S_1 ··· S_{j-1} S_j
T
T_k T_{k-1} ··· T_1 ⊣

Initially, the formula is stored in the array R_i (possibly on an input medium); as it is examined, each character passes through S to the list S_j (right brackets and terminators never get past S). During compilation, symbols are taken from the list S_j, and pass through T to the list T_k. R_0, S_0, and T_0 always contain terminator symbols. A list Q_t is used to store generated pseudocodes.

The flow diagram of Figures 1 and 2 embodies the techniques already described. Box 1 performs initialization. Box 2 reads a character from the list R_i. Boxes 3, 4, 5 and 16 assemble the letters of identifiers and replace each identifier with an internal name. Boxes 6–15 and 17–19 process constants, replacing each with an internal name. Box 20 adds a character at the head of list S_j.

Boxes 21–40 generate coding or pseudocode. Box 23 replaces unary minus signs by the word "neg"; unary plus signs are eliminated. The subroutine WRITE places generated pseudocodes on a list, and constructs a name for each partial result. The subroutine COMPILE creates pseudocodes for the arithmetic operators. Two other subroutines maintain a symbol table and constant pool.

Concatenation of symbols is represented by the ⊕ operator. The word obtained by packing i, j, and k into appropriate fields of a single word will be called (i, j, k). Set membership is indicated as follows: $S \in \{a, b, \text{'c'}\}$ asserts that the symbol S belongs to one of the sets whose descriptive names are a and b, or that S is the symbol c.

A genuinely efficient formula translator requires abilities not present in the algorithm of Figures 1 and 2. First, it should perform during compilation those arithmetic operations depending only upon constants. Second, it should make informed decisions between floating and fixed point representations for each constant and partial result. Third, it should apply effectively the commutative law for addition and multiplication. Fourth, it should recognize equivalences based upon the properties of the minus sign, such as $a-b = -(b-a)$.

The first ability is essentially trivial; inspection of a type indicator in all operands of a pseudocode discloses whether all are constants. If so, the value is computed, stored in a constant pool, and assigned a name; this name then replaces the original subexpression.

The other three abilities require considerable elaboration of the flow chart of Figure 2. Each name of a quantity will consist of four fields: type, index, sign, and mode. The type may be I (identifier), C (constant), Q (partial result), or Q* (subscripted variable). The index preserves the individuality of each name, and indicates the relative location of the named object in one of the tables maintained

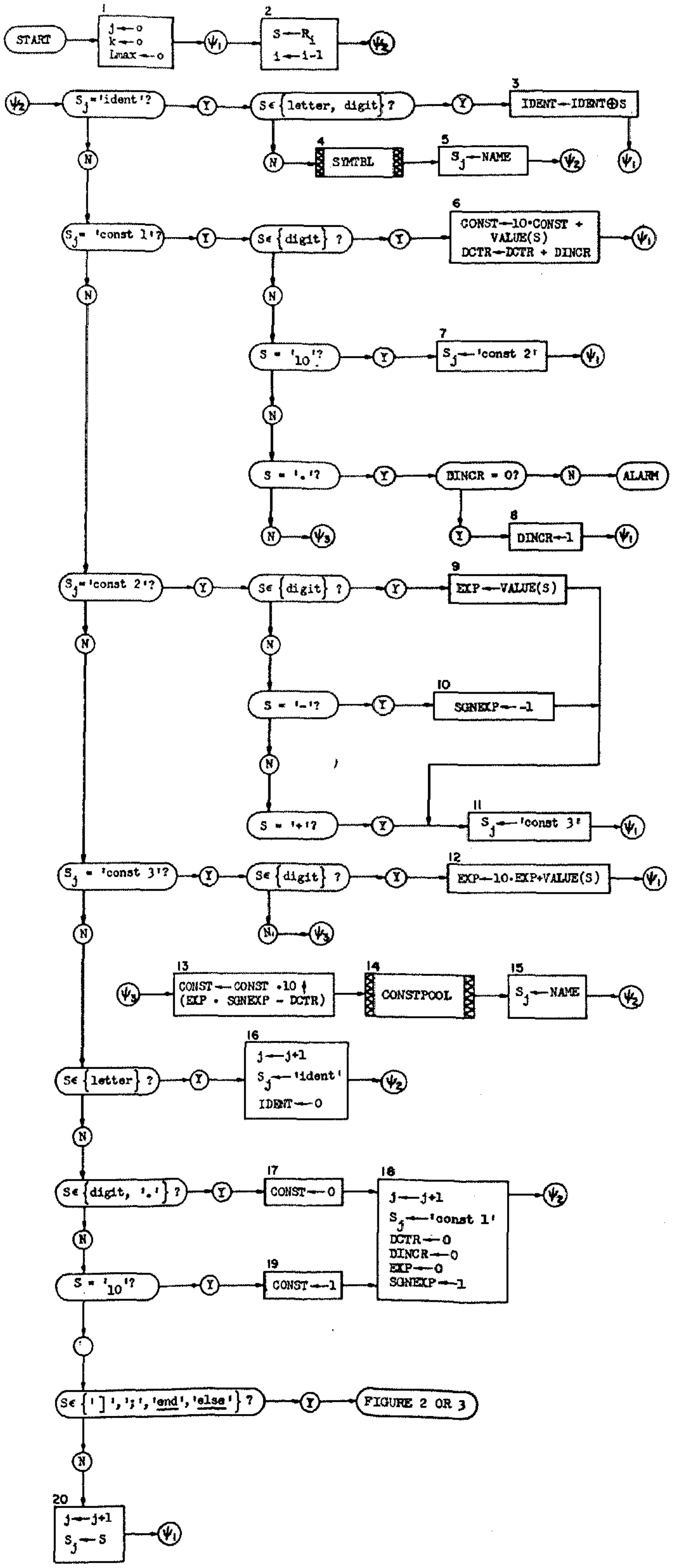


Fig. 1

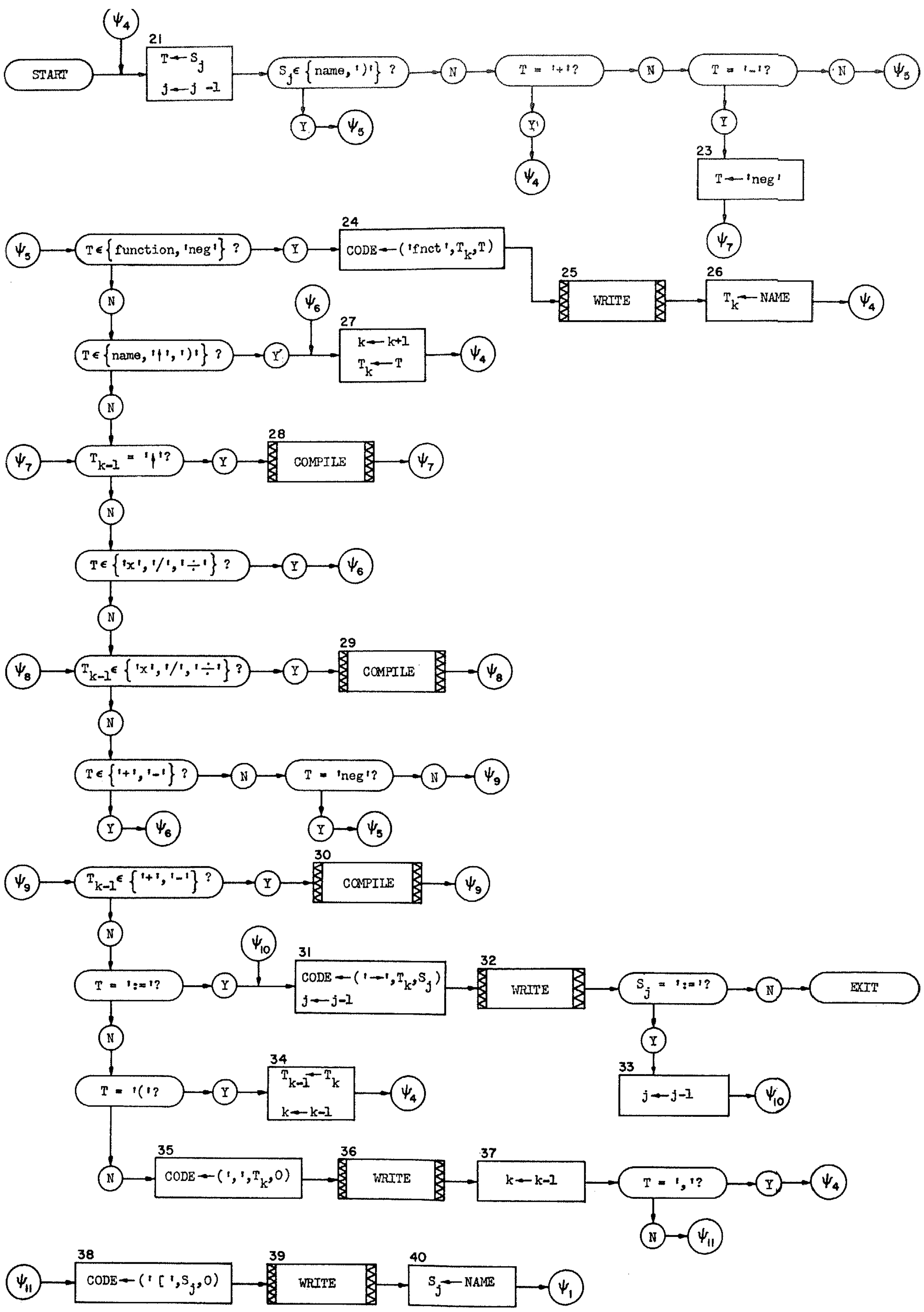

ψ_4
21
START
$T \leftarrow S_j$
$j \leftarrow j-1$
$S_j \in \{$name, ')'$\}$?
N
T = '+'?
N
T = '-'?
N
ψ_5
Y
ψ_5
Y
ψ_4
Y
23
T ← 'neg'
ψ_7
24
ψ_5
$T \in \{$function, 'neg'$\}$?
Y
CODE ← ('fnct', T_k, T)
25
WRITE
26
$T_k \leftarrow$ NAME
ψ_4
N
ψ_6
27
$T \in \{$name, '↑', ')'$\}$?
Y
$k \leftarrow k+1$
$T_k \leftarrow T$
ψ_4
N
28
ψ_7
T_{k-1} = '↑'?
Y
COMPILE
ψ_7
N
$T \in \{$'x', '/', '÷'$\}$?
Y
ψ_6
N
29
ψ_8
$T_{k-1} \in \{$'x', '/', '÷'$\}$?
Y
COMPILE
ψ_8
N
$T \in \{$'+', '-'$\}$?
N
T = 'neg'?
N
ψ_9
Y
ψ_6
Y
ψ_5
30
ψ_9
$T_{k-1} \in \{$'+', '-'$\}$?
Y
COMPILE
ψ_9
N
ψ_{10}
31
32
T = ':='?
Y
CODE ← ('←', T_k, S_j)
$j \leftarrow j-1$
WRITE
S_j = ':='?
N
EXIT
N
Y
34
33
T = '('?
Y
$T_{k-1} \leftarrow T_k$
$k \leftarrow k-1$
ψ_4
$j \leftarrow j-1$
ψ_{10}
35
36
37
N
CODE ← (',', T_k, 0)
WRITE
$k \leftarrow k-1$
T = ','?
Y
ψ_4
N
ψ_{11}
38
39
40
ψ_{11}
CODE ← ('[', S_j, 0)
WRITE
$S_j \leftarrow$ NAME
ψ_1

FIG. 2a

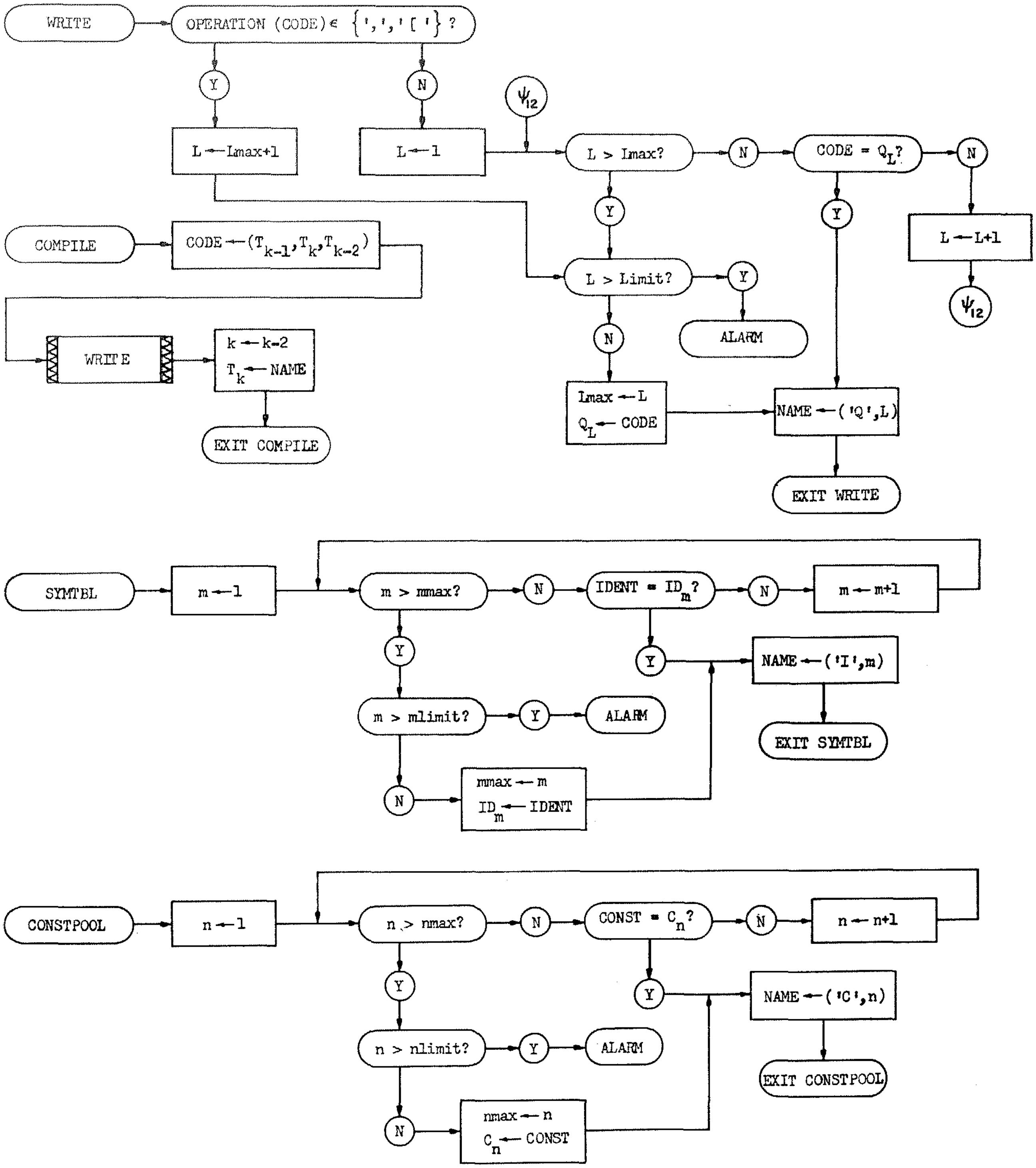


FIG. 2b

by the compiler. The sign bit allows the compiler to name the negative of any quantity which may be named; a one in the sign bit designates the operation of taking the negative. The mode bit distinguishes between fixed and floating point (one and zero, respectively).

Each pseudocode consists of four fields, also: operation, mode, operand 1, and operand 2. The mode bit distinguishes between fixed and floating point arithmetic operations. The operands each consist of the type and index fields of some name. It is assumed that masking operations allow addressing of individual fields within words; for example, x ← type (y) or mode (x) ← 0.

To obtain coding with a minimum of mode conversions, it is important to let the mode of most constants depend upon the context within which they are used. Assuming that the value of a constant has a fixed point representation, it should be treated in fixed point if it is connected by an arithmetic operator to a fixed point variable or partial result. If it is connected to a floating point variable or partial result, it should be treated in floating point. If it

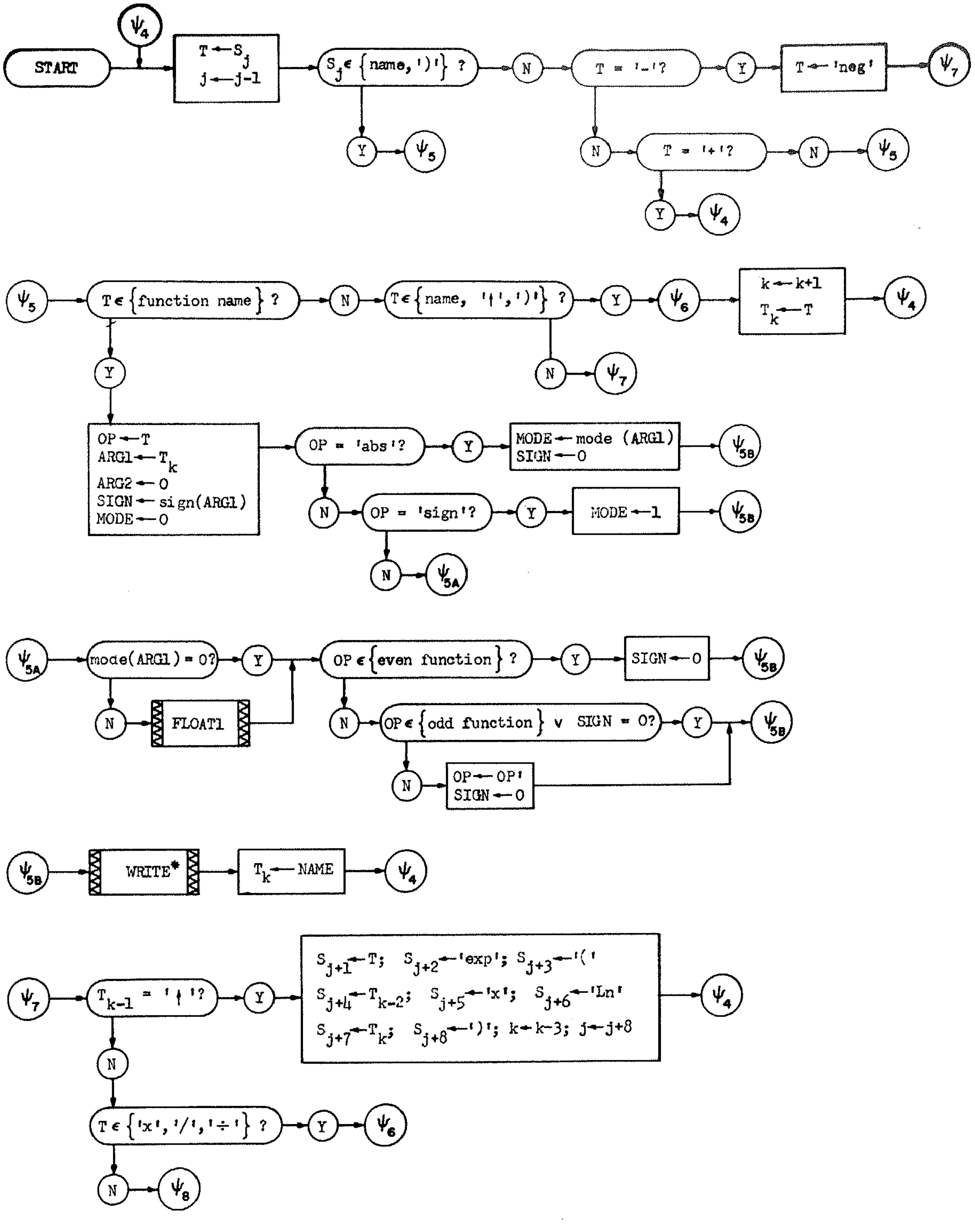


Fig. 3a

is connected to another constant, the value of the expression should be obtained by the compiler, and the whole treated as a single constant. Needless to say, a constant should never be fixed or floated at execution time.

For operators satisfying a commutative law it is possible to reorder the operands in a pseudocode. If a canonical ordering is defined for all names, and each pseudocode for addition and multiplication is written with the operands in correct order, then all subexpressions which may be shown equivalent by repeated application of the commutative laws will be recognized as equivalent by the translator, and coded only once.

For a machine with an accumulator, a canonical ordering should be so chosen that the name of the previous result precedes any other possible operand of a pseudo-code. For multiple-address machines without index registers, in order

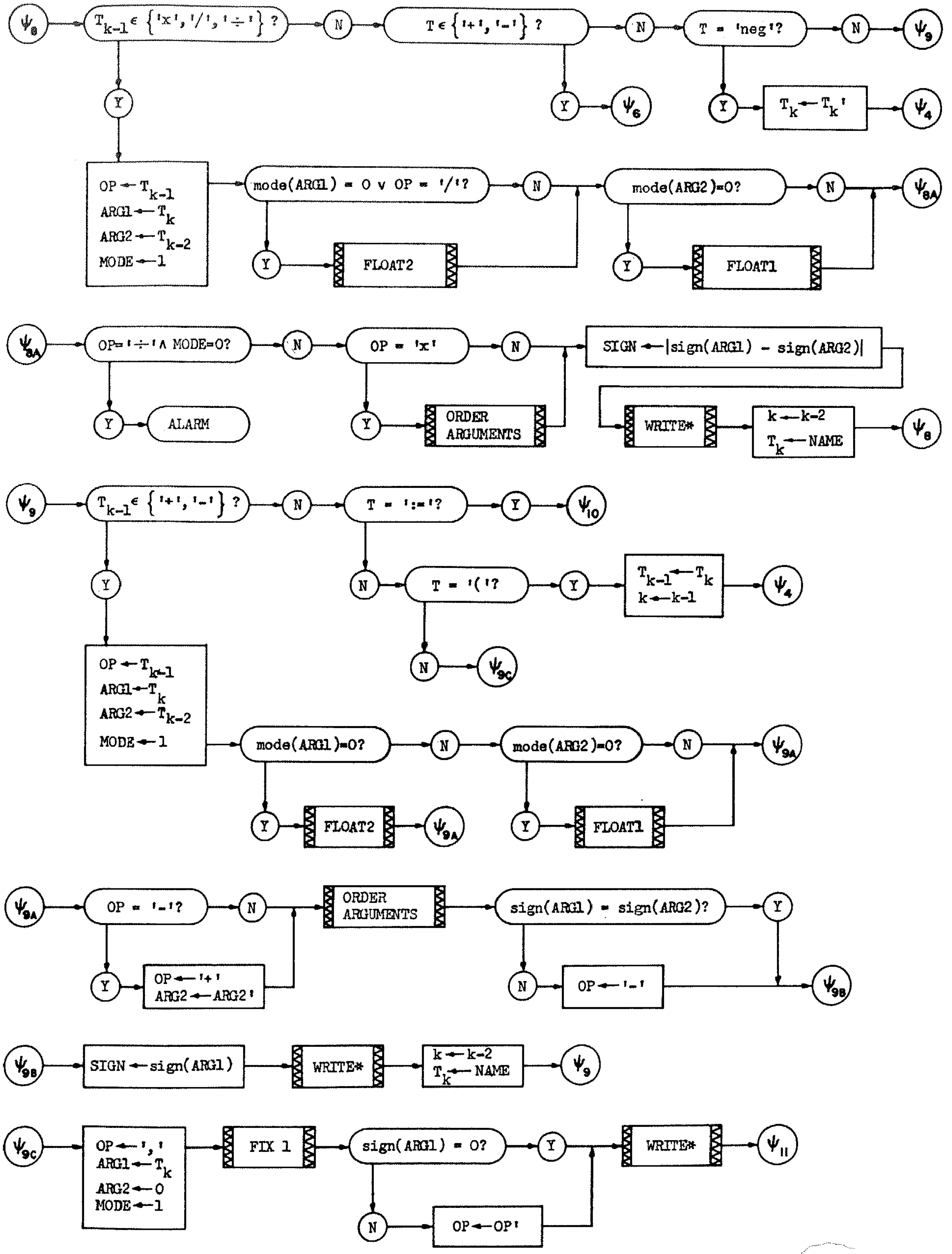


Fig. 3b

to minimize shifting, a modified address (i.e., the immediate result of a subscript operation) should never precede any other type of address. One possible ordering among types is Q < I < C < Q*; the ordering within a given type is the reverse of the numerical order of the indices, and is independent of mode or sign.

The use of a sign bit in each name extends the commutative law; for example, if a prime denotes the operation of complementing the sign bit, $a-b = a+b' = b'+a = (b+a')' = (b-a)'$. Having computed $b-a$, then the compiler should recognize that it need not compute $a-b$. A

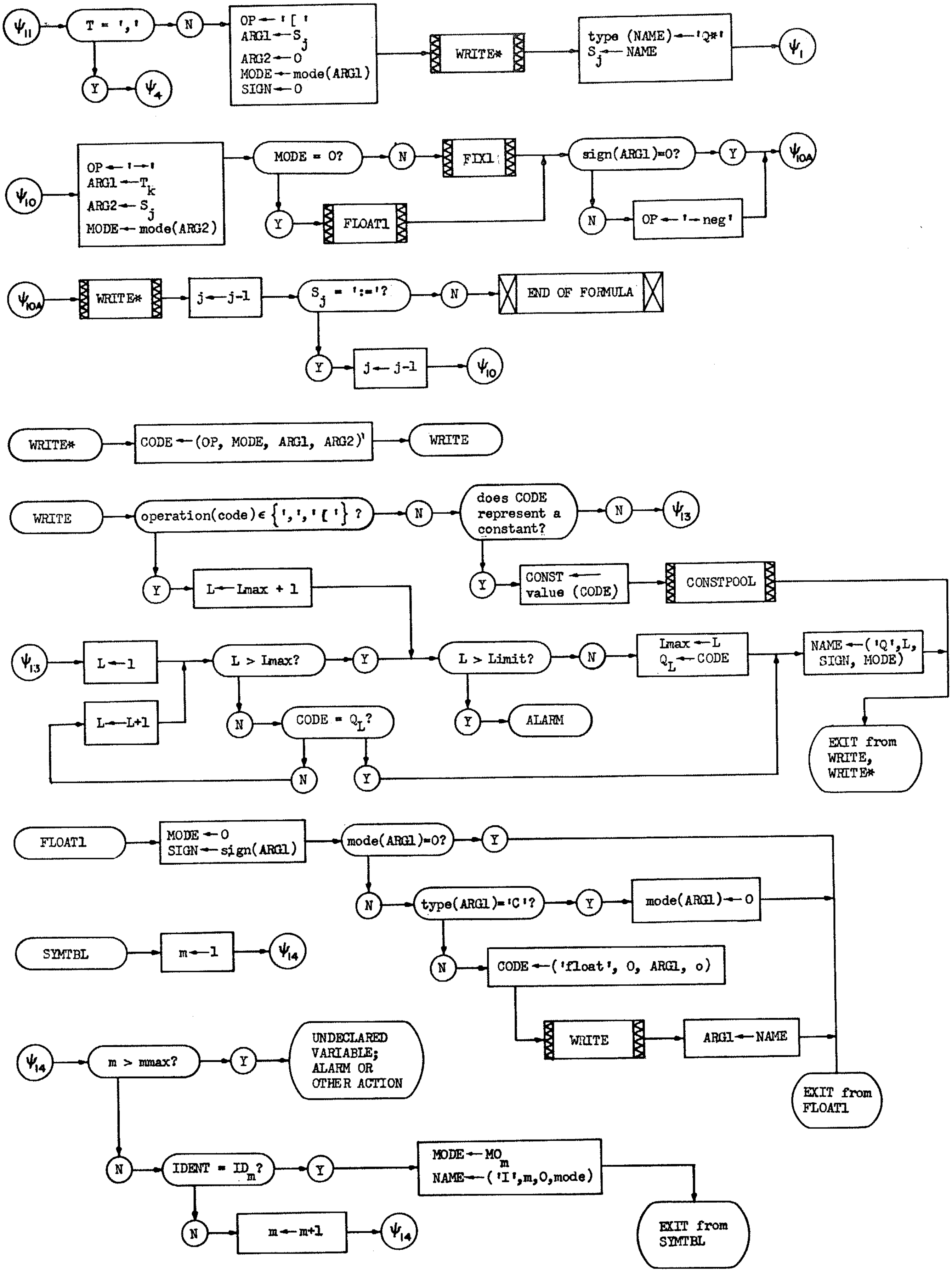

ψ11
T = ','
N
Y
ψ4
OP ← '['
ARG1 ← S_j
ARG2 ← 0
MODE ← mode(ARG1)
SIGN ← 0
WRITE*
type (NAME) ← 'Q*'
S_j ← NAME
ψ1
ψ10
OP ← '←'
ARG1 ← T_k
ARG2 ← S_j
MODE ← mode(ARG2)
MODE = 0?
N
FIX1
Y
FLOAT1
sign(ARG1)=0?
Y
ψ10A
N
OP ← '←neg'
ψ10A
WRITE*
j ← j-1
S_j = ':='?
N
END OF FORMULA
Y
j ← j-1
ψ10
WRITE*
CODE ← (OP, MODE, ARG1, ARG2)'
WRITE
WRITE
operation(code) ∈ {',', '['} ?
N
does CODE represent a constant?
N
ψ13
Y
L ← Lmax + 1
Y
CONST ← value (CODE)
CONSTPOOL
ψ13
L ← 1
L > Lmax?
Y
L > Limit?
N
Lmax ← L
Q_L ← CODE
NAME ← ('Q',L, SIGN, MODE)
L ← L+1
N
CODE = Q_L?
Y
ALARM
N
Y
EXIT from WRITE, WRITE*
FLOAT1
MODE ← 0
SIGN ← sign(ARG1)
mode(ARG1)=0?
Y
N
type(ARG1)='C'?
Y
mode(ARG1) ← 0
SYMTBL
m ← 1
ψ14
N
CODE ← ('float', 0, ARG1, o)
WRITE
ARG1 ← NAME
ψ14
m > mmax?
Y
UNDECLARED VARIABLE; ALARM OR OTHER ACTION
EXIT from FLOAT1
N
IDENT = ID_m?
Y
MODE ← MO_m
NAME ← ('I',m,0,mode)
N
m ← m+1
ψ14
EXIT from SYMTBL

Fig. 3c

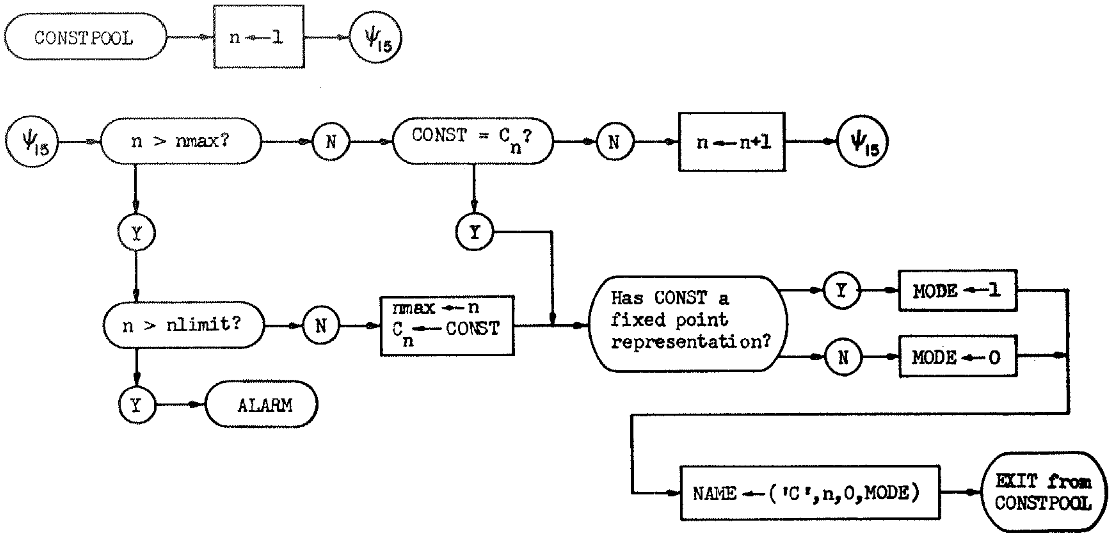


Fig. 3d

typical example of the saving accomplished through use of the sign bits is shown below.

$$w := x - y \times z$$

```
Without sign bit      Using sign bit
   CLA y           CLA y   or   CLA y
   MLY z           MLY z        MLY z
   STR t1          SUB x        SUB x
   CLA x           STN w        CLS accumulator
   SUB t1                       STR w
   STR w
```

The pseudocodes generated by this example, and the successive states of the formula, are shown below:

$w := x - y \times z$	$Q_1 \;:\; \times, y, z$
$w := x - Q_1$	
$w := x + Q_1^-$	
$w := Q_1^- + x$	$Q_2 \;:\; -, Q_1, x$
$w := Q_2^-$	$Q_3 \;:\; \rightarrow \text{neg}, Q_2, w$

Every subtraction a−b is always rewritten as an addition a+b′. The operands may then be placed in the canonical order. The procedure described is particularly efficient upon a machine with a store negative command. Without a store negative command, a more limited use should be made of the sign bit.

Figure 3 presents a compilation algorithm having all the abilities listed above. The rules of decomposition for formulae closely correspond to those used by Figures 1 and 2; in fact, Figure 1 is common to both algorithms. For the remainder, there is a rough correspondence between Figure 2 and Figure 3. Wherever possible, remote connectors in corresponding positions have been given identical names.

The subroutine FLOAT 1 has been added to the process to convert the first operand of a pseudocode to floating point where necessary. Two analogous subroutines, FLOAT 2 and FIX 1 may be diagrammed by simple symbol substitutions in FLOAT 1 and therefore are not shown. The WRITE subroutine now bears the responsibility for detecting arithmetic pseudocodes all of whose operands are constants, evaluating the pseudocode, storing the value in the constant pool, and assigning a name to the result.

The COMPILE subroutine of Figure 2 has disappeared from Figure 3; the arithmetic operations are too thoroughly differentiated in their transformation rules to allow a common compilation sequence for all. The symbol table and constant pool subroutines have been modified to create the new name structures used by Figure 3.

EXAMPLE 1: Application of Figures 1 and 2 to the formula

$$\textbf{begin}\ x := y + z1 \times u\ \textbf{end}$$

```
S ← 'begin'  ;  S1 ← 'begin'
S ← 'x'  ;  S2 ← 'ident'  ;  IDENT ← 0  ;  IDENT ← 'x'
S ← ':='  ;  ID1 ← 'x'  ;  NAME ← 'I1'  ;  S2 ← 'I1'
    S3 ← ':='
S ← 'y'  ;  S4 ← 'ident'  ;  IDENT ← 0  ;  IDENT ← 'y'
S ← '+'  ;  ID2 ← 'y'  ;  NAME ← 'I2'  ;  S4 ← 'I2'
    S5 ← '+'
S ← 'z'  ;  S6 ← 'ident'  ;  IDENT ← 0  ;  IDENT ← 'z'
S ← '1'  ;  IDENT ← 'z1'
S ← '×'  ;  ID3 ← 'z1'  ;  NAME ← 'I3'  ;  S6 ← 'I3'
    S7 ← '×'
S ← 'u'  ;  S8 ← 'ident'  ;  IDENT ← 0  ;  IDENT ← 'u'
S ← 'end'  ;  ID4 ← 'u'  ;  NAME ← 'I4'  ;  S8 ← 'I4'
```

At this point the S-list contains

begin	I_1	:=	I_2	+	I_3	×	I_4

```
T ← 'I4'  ;  T1 ← 'I4'
T ← '×'  ;  T2 ← '×'
T ← 'I3'  ;  T3 ← 'I3'
T ← '+'  ;  CODE ← ('×', 'I3', 'I4')  ;  Q1 ← ('×', 'I3', 'I4')
   NAME ← 'Q1'  ;  T1 ← 'Q1'  ;  T2 ← '+'
T ← 'I2'  ;  T3 ← 'I2'
T ← ':='  ;  CODE ← ('+', 'I2', 'Q1')  ;  Q2 ← ('+', 'I2', 'Q1')
   NAME ← 'Q2'  ;  T1 ← 'Q2'  ;  CODE ← ('→', 'Q2', 'I1')
   Q3 ← ('→', 'Q2', 'I1')  ;  NAME ← 'Q3'
```

At this point compilation terminates, the Q-list containing:

Q_1	:	×	I_3	I_4
Q_2	:	+	I_2	Q_1
Q_3	:	→	Q_2	I_1

EXAMPLE 2: Application of Figures 1 and 3 to the formula

begin x[i × j] := y − z + 1.3/(z−y) **end**

It is assumed that all identifiers have been declared, and are stored in the symbol table as follows:

ID_1	:	i	(mode = 1)
ID_2	:	j	(mode = 1)
ID_3	:	x	(mode = 0)
ID_4	:	y	(mode = 0)
ID_5	:	z	(mode = 0)

```
S ← 'begin' ; $S_1$ ← 'begin'
S ← 'x' ; $S_2$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'x'
S ← '[' ; NAME ← '$I_3$' ; $S_2$ ← '$I_3$' ; $S_3$ ← '['
S ← 'i' ; $S_4$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'i'
S ← '×' ; NAME ← '$I_1$' ; $S_4$ ← '$I_1$' ; $S_5$ ← '×'
S ← 'j' ; $S_6$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'j'
S ← ']' ; NAME ← '$I_2$' ; $S_6$ ← '$I_2$'
  T ← '$I_2$' ; $T_1$ ← '$I_2$'
  T ← '×' ; $T_2$ ← '×'
  T ← '$I_1$' ; $T_3$ ← '$I_1$'
  T ← '[' ; OP ← '×' ; ARG1 ← '$I_1$' ; ARG2 ← '$I_2$'
      MODE ← 1
    ARG1 ← '$I_2$' ; ARG2 ← '$I_1$' ; SIGN ← 0
        CODE ← ('×', 1, '$I_2$', '$I_1$')
    $Q_1$ ← ('×', 1, '$I_2$', '$I_1$') ; NAME ← '$Q_1$' ; $T_1$ ← '$Q_1$'
        OP ← ',' ;
    ARG1 ← '$Q_1$' ; ARG2 ← 0 ; MODE ← 1
        CODE ← (',', 1, '$Q_1$', 0)
    $Q_2$ ← (',', 1, '$Q_1$', 0) ; NAME ← '$Q_2$' ; OP ← '['
        ARG1 ← '$I_3$'
    ARG2 ← 0 ; MODE ← 0 ; CODE ← ('[', 0, '$I_3$', 0)
    $Q_3$ ← ('[', 0, '$I_3$', 0) ; NAME ← '$Q_3$' ; NAME ← '$Q_3^*$'
        $S_2$ ← '$Q_3^*$'
S ← ':=' ; $S_3$ ← ':='
S ← 'y' ; $S_4$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'y'
S ← '−' ; NAME ← '$I_4$' ; $S_4$ ← '$I_4$' ; $S_5$ ← '−'
S ← 'z' ; $S_6$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'z'
S ← '+' ; NAME ← '$I_5$' ; $S_6$ ← '$I_5$' ; $S_7$ ← '+'
S ← '1' ; S ← 'const1' ; DCTR ← 0 ; DINCR ← 0
    EXP ← 0 ; CONST ← 1
S ← '.' ; DINCR ← 1
S ← '3' ; CONST ← 13 ; DCTR ← 1
S ← '/' ; CONST ← 1.3 ; $C_1$ ← 1.3 ; MODE ← 0
    NAME ← '$C_1$' $S_8$ ← '$C_1$' ; $S_9$ ← '/'
S ← '(' ; $S_{10}$ ← '('
S ← 'z' ; $S_{11}$ ← 'ident'; IDENT ← 0 ; IDENT ← 'z'
S ← '−' ; NAME ← '$I_5$' ; $S_{11}$ ← '$I_5$' ; $S_{12}$ ← '−'
S ← 'y' ; $S_{13}$ ← 'ident' ; IDENT ← 0 ; IDENT ← 'y'
S ← ')' ; NAME ← '$I_4$' ; $S_{13}$ ← '$I_4$' ; $S_{14}$ ← ')'
S ← 'end'
```

At this point the S-list contains

begin	Q_3^*	:=	I_4	−	I_5	+	C_1	/	(	I_5	−	I_4	)

```
T ← ')' ; $T_1$ ← ')'
T ← '$I_4$'; $T_2$ ← '$I_4$'
T ← '−' ; $T_3$ ← '−'
T ← '$I_5$' ; $T_4$ ← '$I_5$'
T ← '(' ; OP ← '−' ; ARG1 ← '$I_5$' ; ARG2 ← '$I_4$'
    MODE ← 1
  MODE ← 0 ; SIGN ← 0 ; OP ← '+' ; ARG2 ← '$I_4^-$'
  OP ← '−' ; SIGN ← 0 ; CODE ← ('−', 0, '$I_5$', '$I_4$')
  $Q_4$ ← ('−', 0, '$I_5$', '$I_4$') ; NAME ← '$Q_4$' ; $T_2$ ← '$Q_4$'
      $T_1$ ← '$Q_4$'
T ← '/' ; $T_2$ ← '/'
T ← '$C_1$' ; $T_3$ ← '$C_1$'
T ← '+' ; OP ← '/' ; ARG1 ← '$C_1$' ; ARG2 ← '$Q_4$'
    MODE ← 1
  MODE ← 0 ; SIGN ← 0 ; CODE ← ('/', 0, '$C_1$', '$Q_4$')
  $Q_5$ ← ('/', 0, '$C_1$', '$Q_4$') ; NAME ← '$Q_5$' ; $T_1$ ← '$Q_5$'
      $T_2$ ← '+'
T ← '$I_5$' ; $T_3$ ← '$I_5$'
T ← '−' ; $T_4$ ← '−'
T ← '$I_4$' ; $T_5$ ← '$I_4$'
T ← ':=' ; OP ← '−' ; ARG1 ← '$I_4$' ; ARG2 ← '$I_5$'
    MODE ← 1
  MODE ← 0 ; OP ← '+' ; ARG2 ← '$I_5^-$' ; ARG1 ← '$I_5^-$'
  ARG2 ← '$I_4$' ; OP ← '−' ; SIGN ← 1
      CODE ← ('−', 0, '$I_5$', '$I_4$')
  NAME ← '$Q_4^-$' ; $T_3$ ← '$Q_4^-$' ; OP ← '+' ; ARG1 ← '$Q_4^-$'
  ARG2 ← '$Q_5$' ; MODE ← 1 ; MODE ← 0 ; ARG1 ← '$Q_5$'
  ARG2 ← '$Q_4^-$' ; OP ← '−' ; SIGN ← 0
      CODE ← ('−', 0, '$Q_5$', '$Q_4$')
  $Q_6$ ← ('−', 0, '$Q_5$', '$Q_4$') ; NAME ← '$Q_6$' ; $T_1$ ← '$Q_6$'
      OP ← '→'
  ARG1 ← '$Q_6$' ; ARG2 ← '$Q_3^*$' ; MODE ← 0
      CODE ← ('→', 0, '$Q_6$', '$Q_3^*$')
  $Q_7$ ← ('→', 0, '$Q_6$', '$Q_3^*$') ; NAME ← '$Q_7$'
```

At this point compilation terminates, the Q-list containing

Q_1	:	×	1	I_2	I_1
Q_2	:	,	1	Q_1	0
Q_3	:	[	0	I_3	0
Q_4	:	−	0	I_5	I_4
Q_5	:	/	0	C_1	Q_4
Q_6	:	−	0	Q_5	Q_4
Q_7	:	→	0	Q_6	Q_3^*

For a typical single address machine, the above pseudocodes might generate the following symbolic coding:

```
CLA     j
MLY     i
STR     IXR1
CLA     z
FSB     y
STR     t1
CLA     const1
FDV     t1
FSB     t1
STR     x0, 1
```

RESTRICTIONS

In the last example, the symbolic coding generated is at least comparable to the results of hand coding. Other examples, however, could disclose the limitations of the algorithm. Its inability to apply the associative laws may result in unnecessary mode conversions and storage of partial results in computing sums or products of quantities of unlike modes. In justification, it may be said that floating-point arithmetic is only approximately associative. Its inability to recognize equivalent subexpressions containing subscripted variables is a more serious drawback, and more nearly intrinsic to the algorithm. Finally, no provision has been made to recognize integral constant exponents. Most existing compilers waste time extravagantly by using exp (2 × ln (x)) to compute x ↑ 2. It is possible to rewrite such expressions to be evaluated by a small number of multiplications. For example, y ↑ 9 may be written

$$((((y \times y) \times (y \times y)) \times ((y \times y) \times (y \times y))) \times (((y)))).$$

The disposition of the parentheses is computed by numbering the multiplication signs consecutively. If n is divisible by 2^k but not by 2^{k+1}, then the $\underline{n}$th multiplication sign is preceded by k right parentheses, and followed by k left parentheses. If the last multiplication sign is numbered m, then the entire expression is surrounded by k parentheses, where $2^k > m$. The extension to negative integral exponents is obvious. The rewritten expressions are compiled in the normal manner, the equivalent subexpressions being automatically recognized.

An operational translator would require additional tests at several points to detect symbol strings not allowed by the language. Such tests are omitted here for the sake of clarity in the flow charts.

ACKNOWLEDGMENT

The author is indebted to Arthur Anger, presently at Harvard University, for many helpful criticisms and suggestions, and for coding the algorithm on the UNIVAC 1105.

REFERENCES

1. ERSHOV: *Programming Programme for the BESM Computer.* Pergamon, 1959.
2. WESGTEIN, J. H. From formulas to computer oriented language. *Comm. ACM 2* (Mar. 1959), 6–8.
3. ARDEN, B., and GRAHAM, R. On GAT and the construction of translators. *Comm. ACM 2* (July 1959), 24–26.
4. KANNER, H. An algebraic translator. *Comm. ACM 2* (Oct. 1959), 19–22.
5. SAMELSON, K., and BAUER, F. L. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76-83.

A Syntax Directed Compiler for ALGOL 60*

Edgar T. Irons†

Princeton University, Princeton, N. J.

Although one generally thinks of a compiler as a program for a computer which translates some object language into a target language, in fact this program also serves to *define* the object language in terms of the target language. In early compilers, these two functions are fused inextricably in the machine language program which is the compiler. This fusion makes incorporation into the compiler of extensions or modifications to the object language extremely difficult.

This paper describes a compiling system which essentially separates the functions of *defining* the language and *translating* it into another. Part 1 presents the meta-language used to define the object language in terms of the target language. This meta-language is an extension of the syntax meta-language used in the ALGOL 60 report which allows specification of meaning (in terms of the target language) as well as of form. This succinct definition allows modifications to the form or meaning of the object language to be incorporated easily into the system, and in fact makes the original specification of the object language a reasonably easy task. Part 2 is a description of the program which utilizes a direct machine representation of the meta-linguistic specifications to effect a translation.

Before proceeding to a description of the meta-language we wish to demonstrate heuristically that the proposed meta-language does suffice to specify a translation for any language it can describe. If one proposes to translate language A into language B, it is necessary to have some kind of description of language A in terms of language B. More specifically, one must be able to describe the alphabet of A in B, and must have a set of rules for assigning meaning in B to various possible structures which can be formed in A by concatenating the characters of A's alphabet. The set of rules might be called the syntax of language A, if one considers definitions (in the usual sense of the word) to be merely additional rules of syntax. A translation process might then be to start with the beginning symbols of the string to be translated and to assign meaning and a new syntactic name to symbol groups as they fall into the several syntactic structures. Having thus formed a new set of syntactic elements, the next step is to modify the meanings or amplify them according to the new structures into which these syntactic elements fall. If one considers the characters of the alphabet to be syntactical units themselves, the two steps in the process are indeed indentical. Evidently the only restriction necessary to make such a description uniquely specify a language is that there be a unique syntactic structure for any possible finite string of symbols in the language.

*This report was supported, in part, by the Office of Naval Research under Contract Nonr-1858(22). Reproduction, translation, publication, use and disposal in whole or in part by or for the United States Government is permitted.

† The work leading to this report was done while the author was employed principally by Princeton University and later by the Communications Division of the Institute for Defense Analyses, Princeton.