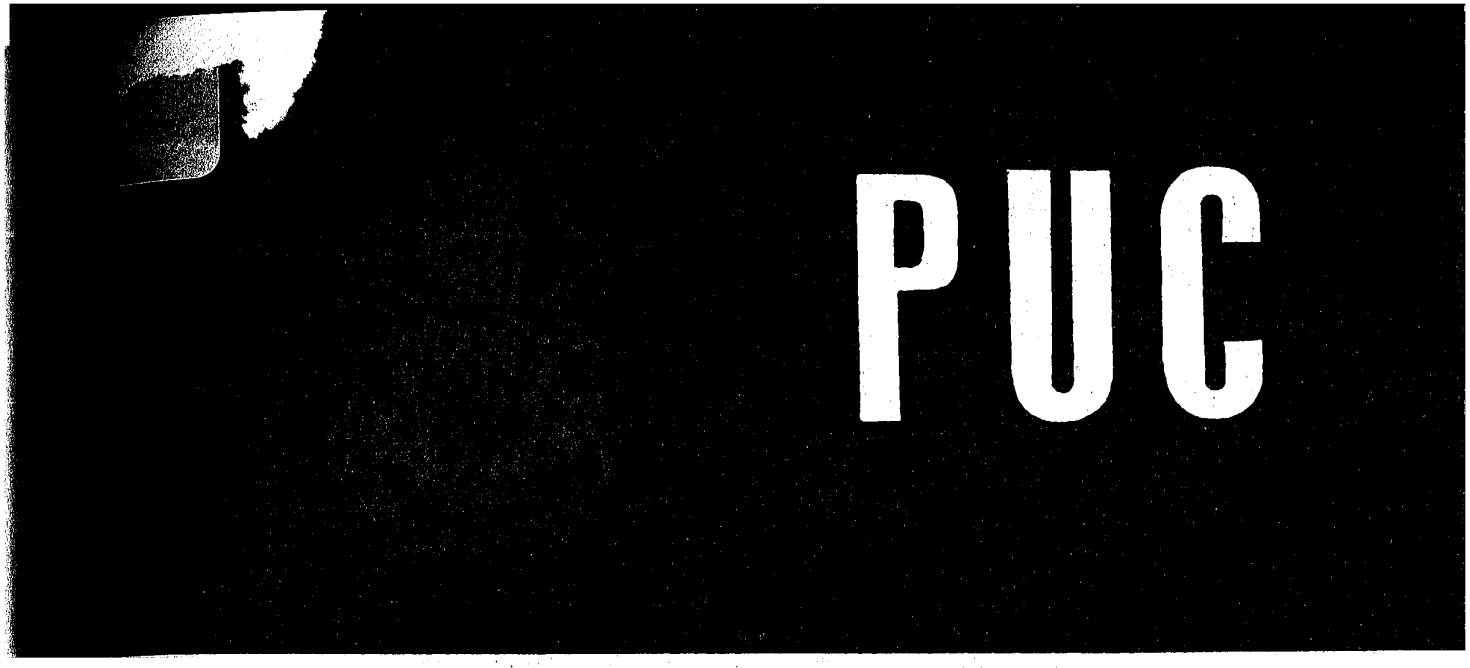


Series: Monographs in Computer Science
and Computer Applications

Nº 3/75

ON THE GENERATION OF IMPROVED INTERMEDIATE LANGUAGE FOR A CLASS OF ARITHMETIC EXPRESSIONS

BY

Sergio E.R. Carvalho

Computer Science Department

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

ON THE GENERATION OF IMPROVED INTERMEDIATE LANGUAGE FOR A CLASS OF ARITHMETIC EXPRESSIONS

Sergio E. R. Carvalho
Associate Professor
Computer Science Department
(Informática) PUC/RJ

Series Editor: Prof. S.M. dos Santos

April/ 1975

ABSTRACT

Many algorithms for the production of improved object code for arithmetic expressions have been described in the literature. In particular, Sethi and Ullman [6] present an algorithm for which the optimality property is proved with respect to the length of code produced. This algorithm consists mainly in modifying the intermediate language representation of an arithmetic expression of a certain class in such a way that full advantage of the algebraic properties of commutativity and associativity is taken.

In this paper we describe an augmented syntactical analyzer that directly produces improved intermediate language in the sense of [6], given an arithmetic expression in the same class as described in [6].

1. INTRODUCTION AND PREVIEW

The generation of improved object code for arithmetic expressions has been studied in some detail. For example, the contributions of Floyd [1], Anderson [2], Gear [3], Nakata [4] and Busam and Englund [5] are among the most important early results obtained in this area. More recently, Sethi and Ullman [6], Cocke and Schwartz [7], Beatty [8] and Earnest [9] have developed new techniques to improve the generation of object code.

In particular, Sethi and Ullman's algorithms (also presented in Aho and Ullman [10]) are shown to really produce optimal object code, with respect to criteria as for instance the length of the code generated, and the number of registers used. This algorithm consists mainly in taking advantage of the commutative and associative algebraic laws, by adequately manipulating a syntax tree (intermediate language) produced by the syntactical analysis phase of a compiler. The class of arithmetic expressions considered in [6,10] is restricted in the sense that:

(i) only one arithmetic expression can be considered at a time;

(ii) the operands are such that the algebraic laws of commutativity and associativity hold.

Here we shall construct an augmented syntactical analyzer and code generation algorithms which will be able to produce directly improved code for expressions in the class described above.

In what follows we shall assume familiarity with the basic notions or compilers and formal language theory. Our notation is the one used in Aho and Ullman [13].

We shall use as the basic model for our augmented analyzer the operator precedence analyzer described in Floyd [11] . In this efficient bottom-up technique, at each step during the analysis of an expression E, a certain substring to be reduced (the handle) is detected, and is then replaced by an unique nonterminal symbol. The handle detected corresponds to the leftmost operation that can be performed in the current sentential form. Accompanying this reduction process, a compiler usually constructs an intermediate representation of the given expression E. In this paper we shall consider this intermediate representation to consist of a syntax tree [13].

Instead of detecting handles in the usual way, we shall have our augmented analyzer detecting "extended handles". These shall be defined formally in section 2. Briefly they consist of a handle which is extended (in general to the right and to the left) in such a way that each extension corresponds to a "future handle". A corresponding syntax tree is generated only after a handle can be no further extended. In this generation we shall use directly the properties of commutativity and associativity.

In section 3 we describe our algorithm, which consists basically in an iteration of the following steps:

(i) detect a handle;
(ii) extend the handle detected;
(iii) generate the corresponding syntax tree;
(iv) replace the extended handle by a reference to the corresponding syntax tree, until the given arithmetic expression is completely analyzed.

2. BASIC DEFINITIONS

In what follows we shall restrict ourselves to the case in which the handle detected has a concrete meaning for the code generation phase. In other words, we shall consider only handles which correspond to some algebraic operation on operands of the given arithmetic expression.

Let $G = (N, \Sigma, P, S)$ be a grammar generating our class of arithmetic expressions. Let $\gamma = \alpha\beta_0 w$ (for $\alpha \in (N \cup \Sigma)^*$, $\beta_0 \in (N \cup \Sigma)^+$ and $w \in \Sigma^*$) be a rightmost sentential form (shortly RSF) derived in G, and let $\beta_0 = X\theta Y$ be the handle of γ. Suppose that $A \to \beta_0$ is the only production in P with the nonterminal A on the left side.

Let γ_1 be the RSF obtained from γ by substituting A for β_0. Let us also denote by $\gamma_2, \gamma_3, \ldots, \gamma_n$ the RSF's obtained from the successive replacement of handles.

Definition: We recursively define the k-extended handle of γ, for $k = 0, 1, 2, \ldots$, as follows:
basic step: β_0 is the 0 - extended handle of γ;
induction step: Let β_{k-1} be the $(k-1)$ - extended handle of γ, for $k = 1, 2, \ldots$.

We now distinguish two cases:

CASE 1: θ (the operator in β_0) is not associative:
If we have in γ the configuration $\alpha(\beta_{k-1})w$, then $\beta_k = (\beta_{k-1})$ is the k-extended handle of γ.

CASE 2: θ is associative:

If we have in γ one of the configurations below:

(i) $\alpha(\beta_{k-1})w$;

(ii) $\alpha\ \beta_{k-1}\ \theta_r\ X_r\ w$, where $\theta_r = \theta$ and X_r is an operand;

(iii) $\alpha\ X_\ell\ \theta_\ell\ \beta_{k-1}\ w$, where $\theta_\ell = \theta$ and X_ℓ is an operand

and if β_k is the handle of γ_k, where β_k is either

(i) $\beta_k = (\beta_{k-1})$

or

(ii) $\beta_k = \beta_{k-1}\ \theta_r\ X_r$

or

(iii) $\beta_k = X_\ell\ \theta_\ell\ \beta_{k-1}$,

then β_k is the k-extended handle of γ .

Definition: Let k be such that there is no (k+1) - extended handle of γ . Then β_k is the extended handle of γ and is denoted by β_E.

To illustrate the definitions above, consider the grammar G given by the productions below. This grammar will be our generating grammar throughout the paper.

S → ⊢ E ⊣
E → T | E + T | E - T
T → F | T * F | T / F
F → I | (E)

We shall now impose a restriction in L(G) of semantical nature, in order that the arithmetic expressions generated by G may belong to the class defined above: each instance of the identifier class I must be distinct.

Example 2.1. Suppose now we are given the expression ⊢(a + b + c) + ((d * (e * j))/g)⊣ . The Table below shows the successive handles and extended handles detected.

handle	extended handle	replaced by
a + b	(a + b + c)	t1
e * f	(d * (e * f))	t2
t2 / g	(t2 / g)	t3
t1 + t3	T1 + t3	t4
⊢ t4 ⊣	-	-

Note that the replacement of each instance of an identifier by I is neglected above, in accordance with a previous restriction.

Before proceeding to introduce our augmented syntactical analyzer, let us examine extended handles more closely. Given the arithmetic expression E, let T_E be the corresponding syntax tree. First it should be noted that a nontrivial (case 2 in the definition) extended handle of E corresponds to a subtree T' of T_E, such that all its operator nodes correspond to the same operator, which is by definition associative. It follows that when T' is being generated, one can make immediate use of associativity, optimizing in this way a part of the syntax tree T_E.

If the current extended handle is as defined in case 1, then only commutativity can in some cases be applied. Applying these informal notions to example 2.1, we see that initially we have the extended handle (a + b + c). In this case the corresponding syntax tree is a shown in Fig.1.

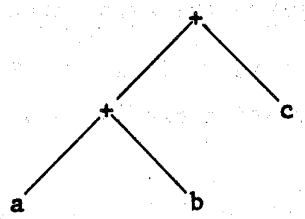


FIG. 1.

The tree above expresses the order of evaluation exactly as is done in the corresponding extended handle: clearly in this case not much is gained by finding the extended handle.

The next extended handle is (d * (e * f)). The corresponding syntax tree to be generated is shown in Fig. 2.

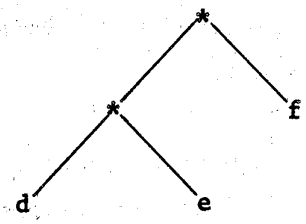


FIG. 2.

Here it can be seen that the order of evaluation explicit in the tree is not the same as the one in the extended handle. It can be shown [6,10] that this use of associativity improves the object code generated.

3. THE GENERATION OF IMPROVED INTERMEDIATE LANGUAGE

In this section we describe our augmented syntactical analyzer. We do so by using a simple language for structured programming. Statements in this language are similar to Pascal statements [12], with the distinction that special delimiters are added. The program below is constructed in several levels: level 1 is the higher, and lower level descriptions are indicated by the occurrence of the keyword run followed by the name of the subprogram to be developed. For clarity, we shall assume as "primitives" many operations that in an actual implementation would have to be developed in more detail.

Algorithm for the generation of improved intermediate language

Input: a (perhaps not well formed) arithmetic expression E satisfying the restrictions mentioned in section 1.

Output: an error condition, in case E is not well formed, or an improved syntax tree T.

Notation: We use $X_1, X_2, \ldots, Y_1, Y_2, \ldots, X_\ell, X_r$ to denote either identifiers or the unique numbers assigned to already generated subtrees. $\theta, \theta_\ell, \theta_r$ denote operators (+, -, *, /). The capital letters C and A denote the properties of commutativity and associativity, respectively. Comments are enclosed in brackets "{" and "}".

Description:

```
Level 1: begin
         repeat
              2: run "find handle β0; if error, stop";
             if β0 = X1 θ X2 then {handle is semantically meaningful}
               if θ not A then {handle extended only if enclosed in
                                parenthesis}
                 begin
                    3: run "extend 1"; {include parenthesis,if possible}
                   if θ C then {apply commutativity if necessary}
                        4: run "commute"
                   fi
                    5: run "generate 1" {generate corresponding syntax
                                         tree}
                 end
               else {in this case θ is considered to be C. and A.}
                 begin
                      6: run "extend 2"; {extend to left and right, if
                                          possible}
                      7: run "generate 2" {generate corresponding subtree}
                 end
               fi
             fi
              8: run "replace extended handle by unique number of corre-
                      sponding subtree"
         until RSF = ⊢ t ⊣ tpr {exit from the repeat loop only when E is
                                completely analyzed}
         end.
```

Level 2 shall not be refined. We assume the existence of an operator precedence syntactical analyzer that, when a RSF is input, finds the handle or detects an error (case in which the execution is halted).

We now refine level 3.

Level 3: newhandle := β_0;

```
while RSF = α (newhandle) w do {extend newhandle to contain
                                enclosing brackets}
newhandle := (newhandle)
od
```

On exit from the while loop no further extensions are possible.

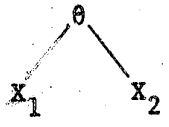
Level 4 is now developed.

Level 4: {extended handle is of the form $({}^{k}X_1\ \theta\ X_2)^{k}$, for $k \geq 0$}

```
if X1 = variable and X2 = number of generated subtree then {com-
                                                    mute operands}
        newhandle := X2θ X1
fi
```

The order of the operands is important in the generation of optimized code, as shown in [6,10].

Level 5: {generation of the syntax tree for the two operands case}

run "construct the tree  " ;

run "assign unique number to tree constructed" ;

No further refinements are necessary from level 5, since the meaning of the two run statements above is clear.

We now develop level 6. In this section of the algorithm nontrivial extensions are performed.

```
Level 6: newhandle := β0;

repeat {extend handle according to case 2 of the definition}
  if RSF = α (newhandle) w then {absorb parenthesis}
    begin
      newhandle := (newhandle);
      possible := true {possible is set to false when no further
                        extensions can be made}
    end
  else
    if RSF = α θi Xℓ θℓ newhandle θr w and θℓ= θ and θi< θℓ and
       θℓ> θr and Xℓ = operand then
      begin
        newhandle := Xℓ θℓ newhandle; {extend to the left}
        possible := true
      end
    else
      if RSF = α θℓ newhandle θr Xr θ2 w and θr = θ and θℓ< θr and
         θr> θ2 and Xr = operand then
        begin
          newhandle := newhandle θr Xr; {extend to the right}
          possible := true
        end
      else possible := false {no extensions possible in this iter-
                              ation}
    fi
  fi
until not possible tpr
```

Control exits from the loop above when no further extensions are possible.

Before refining level 7, we shall assume the existence of an algorithm performing the functions described in level 8, namely:

(i) create an unique number to designate the subtree just generated;

(ii) replace in the current RSF the extended handle just considered by the number created in (i).

Now we consider the generation of syntax trees corresponding to extended handles in which the operator is associative. Let the operands of the extended handle β_E be $X_1, X_2, \ldots, X_n$, $n \geq 2$, where for $1 \leq i \leq n$, X_i may indicate either a variable or a subtree. Let θ be the operator of β_E. In what follows we say that a <u>subtree T is on θ</u> if θ is the operator of the root of T.

Level 7: if for $1 \leq i \leq n$ X_i = variable then

run "construct the tree of Fig. 3"

else {at least one of the operands denotes a subtree}

if for $k \geq 0$ $\beta_E = ({}^k X_1 \theta\ X_2)^k$ then

if X_1 = variable and X_2 = subtree then {commute operands}

run "construct the tree of Fig. 4"

else

if X_1 = subtree on θ and X_2 = subtree on $\theta' \neq \theta$ then

run "construct the tree of Fig. 5"

else run "construct the tree of Fig. 6"

fi

fi

else {β_E has more than two operands, at least one denoting a subtree}

begin

run "drop the parenthesis off $_E$"; {we know θ to be commutative and associative, so the order of evaluation of β_E does not matter}

run "order the operands of β_E"; {in such a way that $\beta_E = Y_1 \theta\ Y_2 \theta \ldots \theta\ Y_k \theta \ldots \theta\ Y_\ell \theta \ldots \theta\ Y_n$, where: for $1 \leq j \leq k$, Y_j denotes a subtree on $\theta' \neq \theta$;

for $k < j \leq \ell$, Y_j denotes a subtree on θ;

for $\ell < j \leq n$, Y_j denotes a variable.

For $k < j \leq \ell$, let $Z_{j1}, Z_{j2}, \cdots, Z_{jm_j}, \cdots, Z_{jn_j}$ be the operand nodes of the θ nodes of the subtree Y_j, where $Z_{j1}, Z_{j2}, \cdots, Z_{jm_j}$ denote subtrees on $\theta' \neq \theta$}

run "construct the tree of Fig. 7"

end

fi

fi

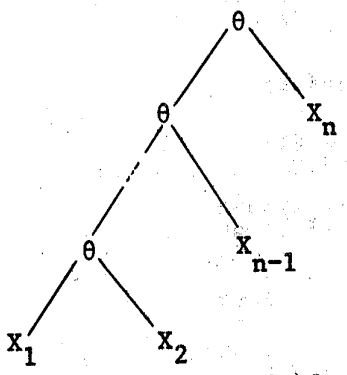


FIG. 3.

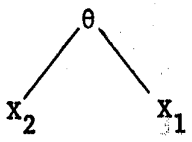
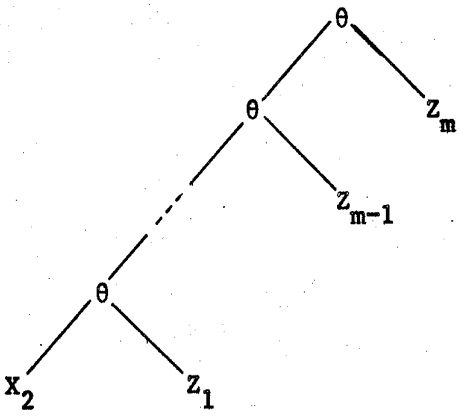


FIG. 4.



where $Z_1, Z_2, \cdots, Z_m$ are the operands of the θ nodes of the subtree denoted by X_1

FIG. 5.

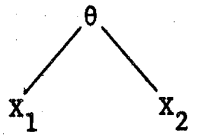
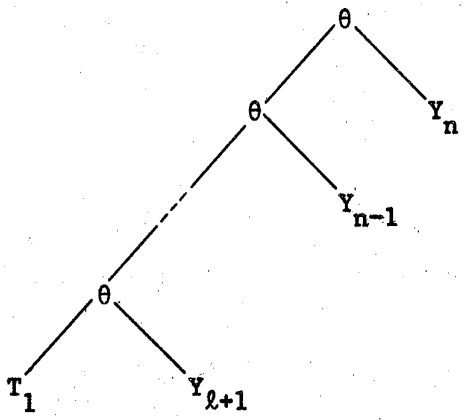


FIG. 6.



where $Y_{\ell+1}, \ldots, Y_{n-1}, Y_n$ are the variable operands of β_E, and T_1 is the subtree of Fig. 8.

FIG. 7.

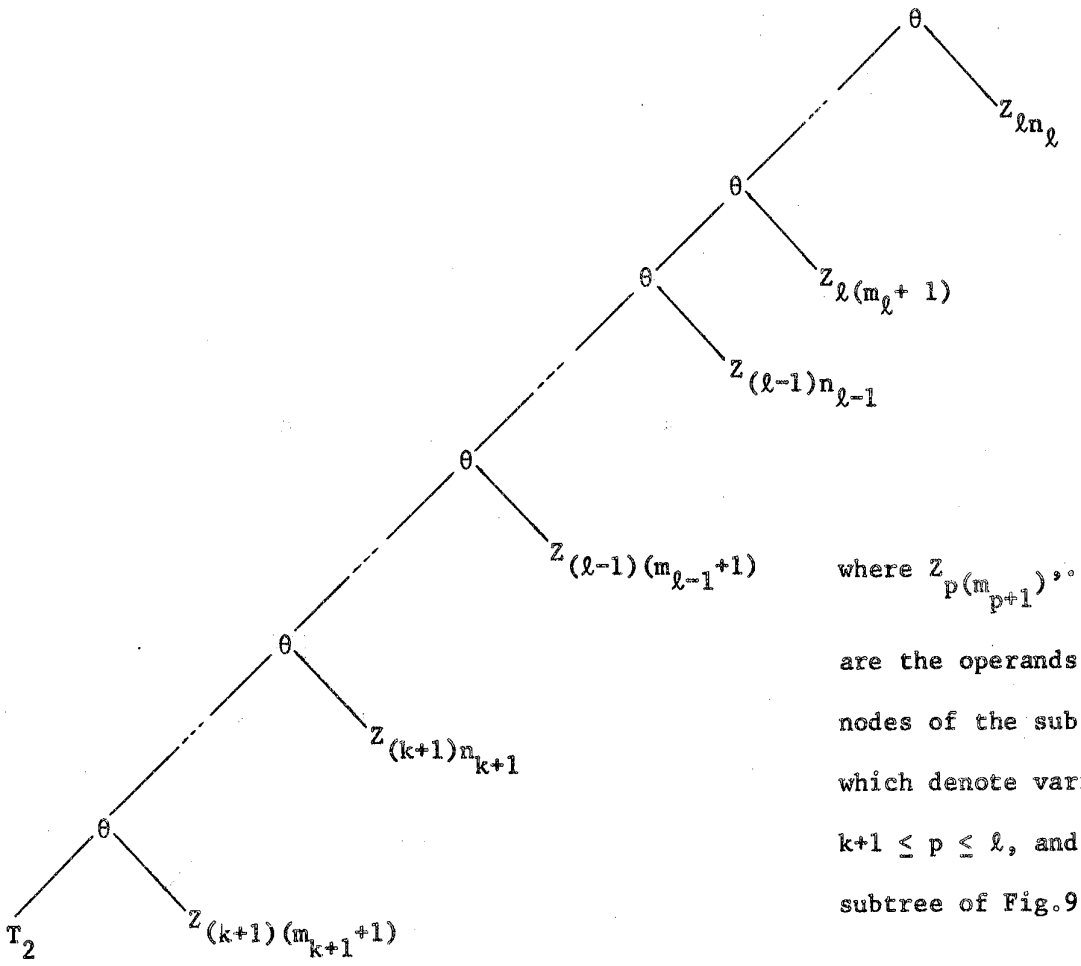


FIG.8.

where $Z_{p(m_{p+1})}, \ldots, Z_{pn_p}$ are the operands of the θ nodes of the subtree Y_p which denote variables, for $k+1 \leq p \leq \ell$, and T_2 is the subtree of Fig.9.

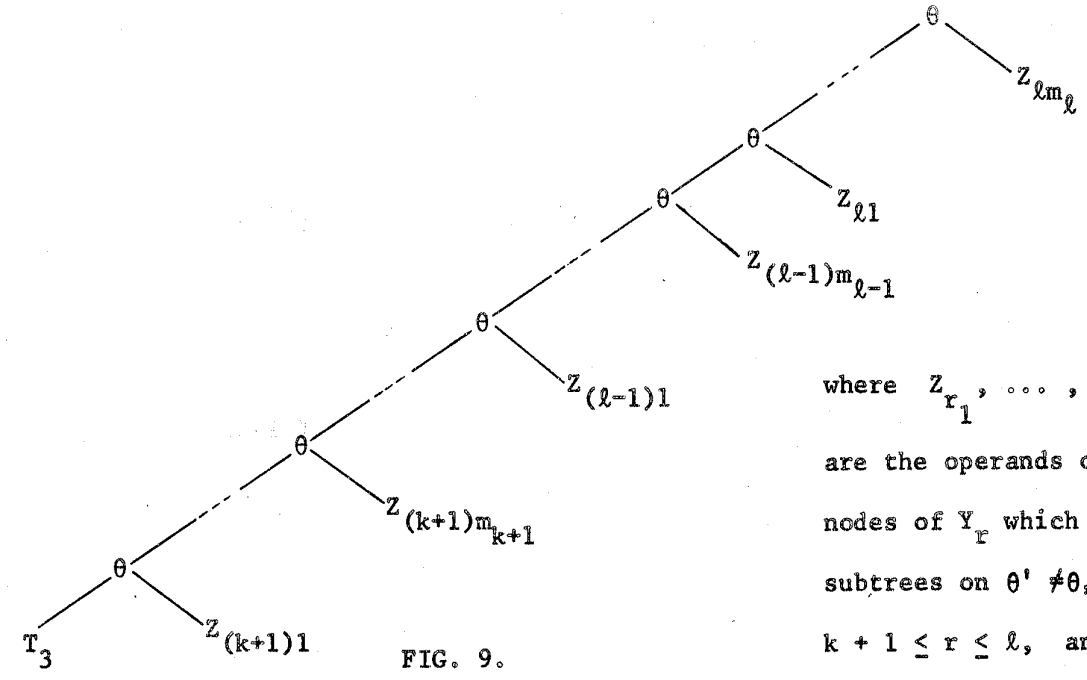
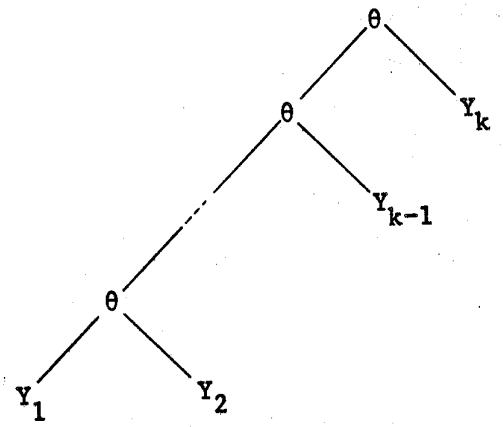


FIG. 9.

where $Z_{r_1}, \ldots, Z_{rm_r}$ are the operands of the θ nodes of Y_r which denote subtrees on $\theta' \neq \theta$, for $k+1 \leq r \leq \ell$, and T_3 is the subtree of Fig. 10.



where $Y_1, \cdots, Y_{k+1}, Y_k$ are the subtrees on $\theta' \neq \theta$ which are operands of β_E.

FIG. 10.

We shall informally discuss the algorithm above. Consider first the level 1, assuming all other levels to be properly refined. Level 1 consists mainly in the repetition of the steps:

(i) detect the current handle (level 2);
(ii) extend the handle, if possible, and generate code;
(iii) reduce the current RSF (level 8).

The algorithm ends when no further reductions are possible in the RSF. Let us now examine step (ii) above. As stated before, only semantically meaningful handles are considered; all others do not generate code and are merely replaced by a number in the RSF. Operators are here considered to be either

(i) not commutative and not associative;
or (ii) commutative but not associative;
or (iii) commutative and associative.

In cases (i) and (ii) the only extension allowed to a handle is the "trivial" one consisting of the addition of the enclosing parenthesis. In case the handle operator is commutative, however, we can in certain cases improve the code generated by switching the operands, [6,10].

In case (iii) extensions to the right and left are permitted under certain conditions. We shall see in the following that when the extended handle operator is both commutative and associative the ordering of the extended handle operands is important in code improvement.

Finally we observe that for each extended handle detected there is a code generation phase.

Consider now level 4. Its input is a substring of the form $({}^{k}X_1\ \theta\ X_2)^{k}$, for $k \geq 0$. We also know θ to be commutative. In this case we commute X_1 and X_2 under exactly the same conditions as stated in [6,10]. Note also that for code generation the parenthesis are not important and are dropped off the extended handle.

The next level deserving further explanation is level 6. This consists of a "repeat" loop in which the exit is a function of the logical variable "possible", which is set to true each time an extension is performed, and is set to false when an iteration is performed without an extension. The loop consists basically in checking the current RSF for the configuration presented in the definition of extended handle. For each possible configuration a corresponding extension is performed.

The core of the algorithm, namely level 7, shall now be explained in more detail. The input to this level is an extended handle in which the operator is commutative and associative. Level 7 consists basically in determining the pattern of the extended handle and in generating code appropriately. The first pattern tested is one in which all operands denote variables. It can be easily seen that the syntax tree generated yields the best possible code. For example, let θ be "+". The code generated in this case could be

```
LOAD   X1
ADD    X2
ADD    X3
 :
 :
ADD    XN
```

Thus to evaluate the extended handle a single accumulator suffices.

In the next pattern at least one operand denotes a subtree (which corresponds to an extended handle detected before). In this case several subpatterns are possible. First the case in chich β_E has only two operands is considered. Here the possibilities are:

(i) X_1 = variable, X_2 = subtree;

(ii) X_1 = subtree, X_2 = variable;

(iii) X_1 = subtree on θ, X_2 = subtree on $\theta' \neq \theta$;

(iv) X_1 = subtree on $\theta' \neq \theta$, X_2 = subtree on $\theta' \neq \theta$

Clearly the case X_1 = variable, X_2 = variable falls in the pattern examined before. It can be easily seen that the above are actually the only possible patterns in the two operands case.

The pattern X_1 = variable, X_2 = subtree is treated in the same way as in [6,10]. Consider now the case in which X_1 = subtree on θ and X_2 = subtree on $\theta' \neq \theta$. The improvement here consists in creating a subtree T for β_E in such a way that advantage is taken from the fact that both T and X_1 are subtrees on θ. Thus we disregard the structure of the subtree denoted by X_1 and consider only the operands of the corresponding θ nodes, creating the tree shown in level 7.

For all other possible patterns in the two operands case we construct the tree of Fig. 6.

It remains to consider the general case. Here β_E has more than two operands and at least one of them denotes a subtree. Due to the algebraic properties of θ, we can reorder the operands of β_E so that the best possible corresponding syntax tree can be constructed. First note that in general operands of an extended handle can be

(i) subtrees on some $\theta' \neq \theta$;

(ii) subtrees on θ;

(iii) variables.

Our first task is to reorder operands so that from left to right they are positioned in the order shown above. We can now begin to construct the syntax tree by creating the tree of Fig. 7.

After constructing the "topmost" part of the tree, we can proceed by considering the operands of the subtrees on θ which are operands of β_E. For each such subtree we consider first the variable operands, and after a new section of the tree having those variables as operands is constructed, we finally consider the group (iii) of operands of β_E, completing the generation.

We now consider examples to illustrate the above.

Example 3.1: Let E be ⊢a * b * c + d *(e * f) + g *(h + i) + j * (k *(ℓ + m) * n⊣ and let +, * be commutative and associative operators.

handle	extension	reordering	syntax tree
a * b	a * b * c	same	t1 (Fig. 1)
e * f	d *(e * f)	d * e * f	t2 (Fig. 2)
t1 + t2	same	same	t3 (Fig. 11)
h + i	(h + i)	h + i	t4 (Fig. 12)
g * t4	same	t4 * g	t5 (Fig. 13)
t3 + t5	same	t5 + t3	t6 (Fig. 14)
ℓ + m	(ℓ + m)	ℓ + m	t7 (Fig. 15)
k * t7	j * (k * t7 * n)	t7 * j * k * n	t8 (Fig. 16)
t6 + t8	same	t8 + t6	t9 (Fig. 17)

The example above is from [10, pg. 896]. It is easy to see that the code generated from the tree t9 has exactly the same length as the one produced from the tree obtained in [10, pg. 901]

Example 3.2: E= ⊢(a * (b - c))*(d *(e * f)) + ((g +(h + i)) +(j + (k + ℓ)))⊣

handle	extension	reordering	syntax tree
b - c	(b - c)	-	t1 (Fig. 18)
a * t1	(a * t1)	t1 * a	t2 (Fig. 19)
e * f	t2 *(d*(e * f))	t2 * d * e * f	t3 (Fig. 20)
h + i	(g + (h + i))	g + h + i	t4 (Fig. 21)
k + ℓ	t3+(t4+(j+(k+ℓ)))	t3+t4+j+k+ℓ	t5 (Fig. 22)

4. CONCLUSIONS

We have shown a way in which the algebraic properties of commutativity and associativity can be used in the generation of improved code for a class of arithmetic expressions. The important feature of the technique just described is that, instead of having to modify syntax trees in order to obtain improved code,one can directly generate improved syntax trees, from where improved code is trivially generated. The immediate consequence of this is that we save one pass in the compilation process. Another possible advantage is expressed by our feeling that the combined cost of the augmented syntactical analyzer and the code generation algorithms is less than the cost of manipulating syntax trees, as done in [6,10].

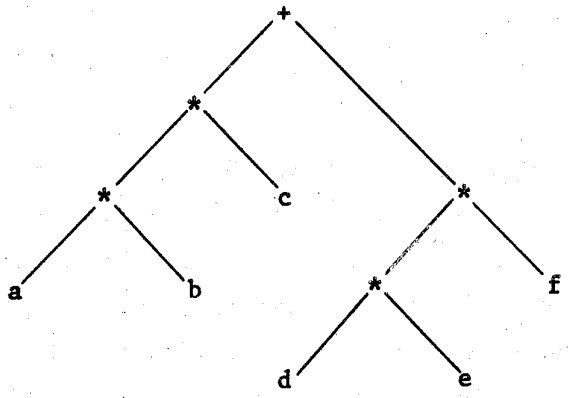


FIG. 11.

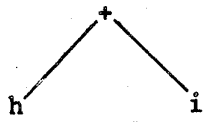


FIG. 12.

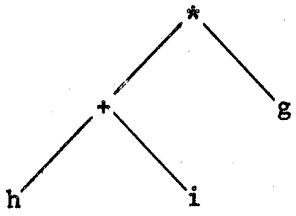


FIG. 13.

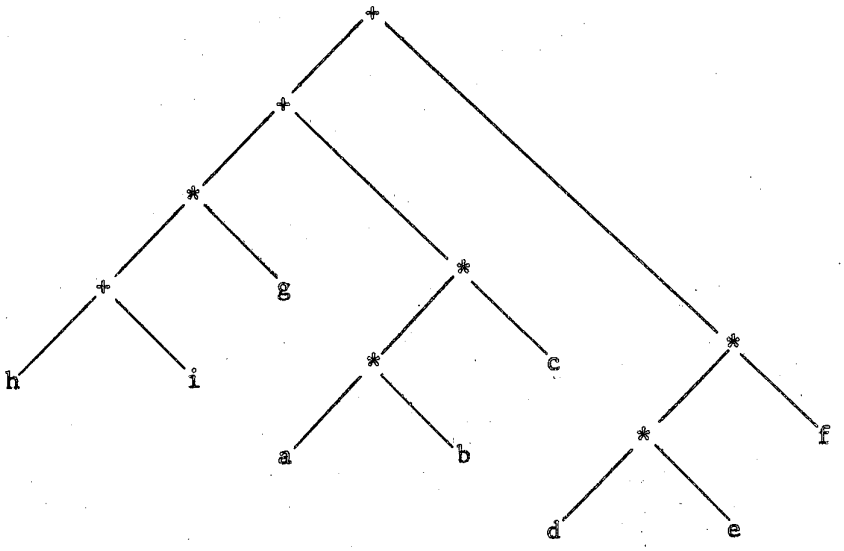


FIG. 14.

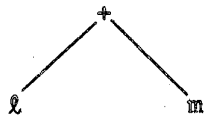
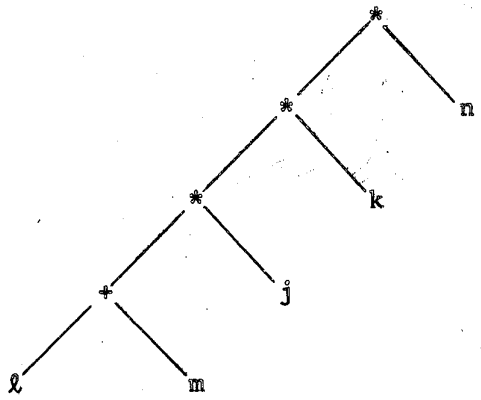


FIG. 15.



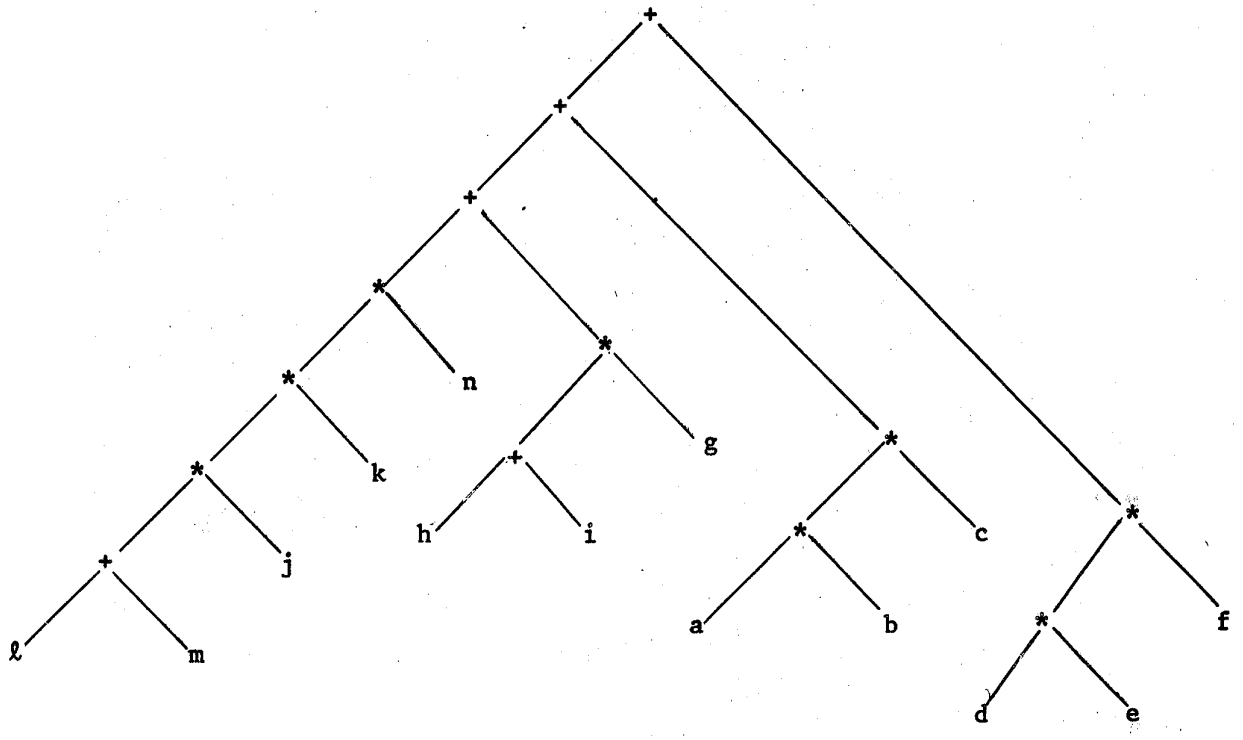


FIG. 17.

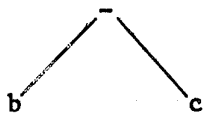


FIG. 18.

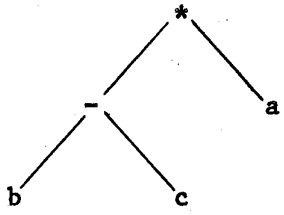


FIG. 19.

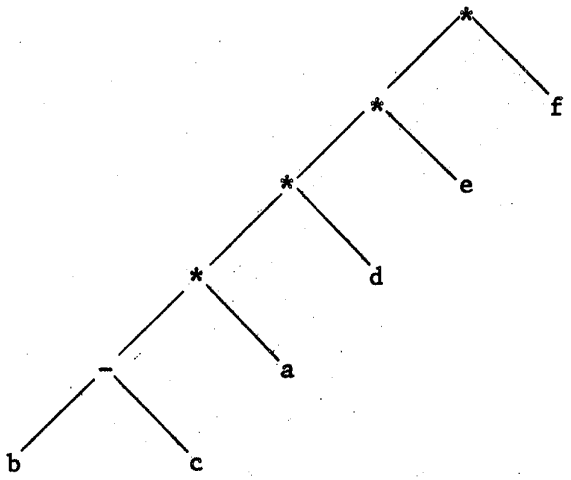


FIG. 20.

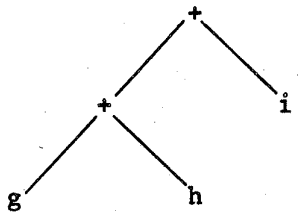


FIG. 21.

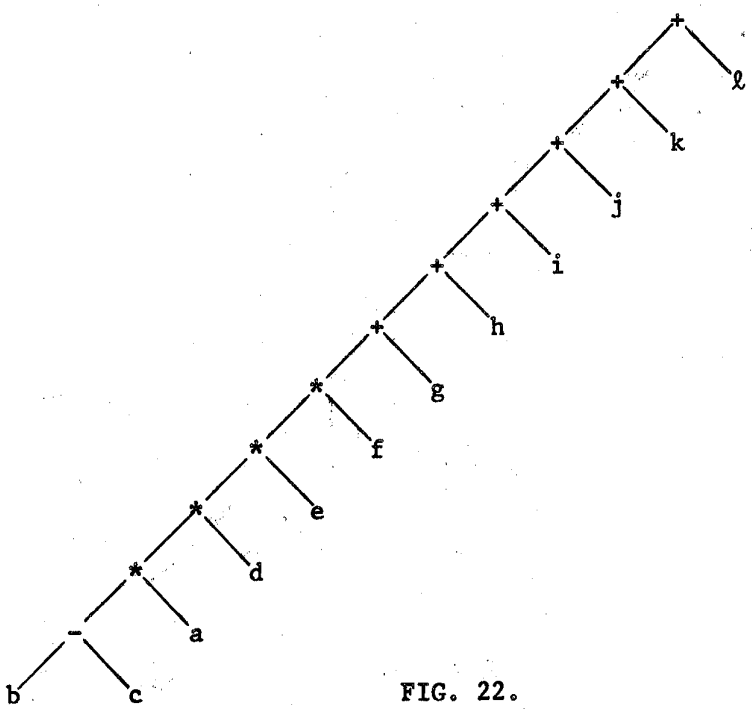


FIG. 22.

BIBLIOGRAPHY

[1] Floyd, R.W. "An algorithm for coding efficient arithmetic operations" Comm. ACM 4:1, 42 - 51, 1961.

[2] Anderson, J.P. "A note on some compiling algorithms" Comm. ACM 7:3, 149 - 150, 1964.

[3] Gear, C.W. "High speed compilation of efficient object code" Comm. ACM 8:8, 483 - 487, 1965.

[4] Nakata, I. "On compiling algorithms for arithmetic expressions" Comm. ACM 12:2, 81 - 84, 1969.

[5] Busam, V.A. & Englund, D.E. "Optimization of expressions in Fortran" Comm. ACM 12:12, 666 - 674, 1969.

[6] Sethi, R. & Ullman, J.D. "The generation of optimal code for arithmetic expressions" J. ACM 17:4, 715 - 728, 1970.

[7] Cocke, J. & Schwartz, J.T. Programming Language and Their Compilers, 2nd edition, Courant Institute of Mathematical Sciences, New York University, New York, 1970

[8] Beatty, J.C. "An axiomatic approach to code optimization for expressions" J.ACM, 19:4, 1972.

[9] Earnest, C. "Some topics in code optimization" J. ACM, 21:1, 76 - 102, 1974.

[10] Aho, A.V. & Ullman, J.D. The Theory of Parsing, Translation and Compiling, Vol. II, Prentice-Hall, 1973.

[11] Floyd, R.W. "Syntactic analysis and operator precedence" J. ACM, 10:3, 316 - 333, 1963.

[12] Wirth, N. "The programming language Pascal (revised report)" Eidgenössische Technische Hoshschule, Zürich, 1972.

[13] Aho, A.V. & Ullman, J.D. The Theory of Parsing, Translation and Compiling, Vol. I, Prentice-Hall, 1972.