The Generation of Optimal Code for Arithmetic Expressions

RAVI SETHI* AND J. D. ULLMAN*

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. The problem of evaluating arithmetic expressions on a machine with $N \geq 1$ general purpose registers is considered. It is initially assumed that no algebraic laws apply to the operators and operands in the expression. An algorithm for evaluation of expressions under this assumption is proposed, and it is shown to take the shortest possible number of instructions. It is then assumed that certain operators are commutative or both commutative and associative. In this case a procedure is given for finding an expression equivalent to a given one and having the shortest possible evaluation sequence. It is then shown that the algorithms presented here also minimize the number of storage references in the evaluation.

KEY WORDS AND PHRASES: arithmetic expressions, associativity, code generation, commutativity, compilers, object-code optimization, register assignment, trees

CR CATEGORIES: 4.12, 5.24, 5.32

1. *Introduction*

Nakata [1] and Meyers [2] consider the problem of minimizing the number of general registers required to evaluate an arithmetic expression. When only commutative operators like + and * are considered, Redziejowski [3] shows that Nakata's algorithm does give the minimum number of registers.

We consider the somewhat more general problem of minimizing the number of program steps and/or the number of storage references in the evaluation of an expression, given a fixed number of general registers. The algorithms we present exploit the commutative and associative properties of operators, and are at the same time able to cope with noncommutative operators. We assume throughout this paper that there are no nontrivial relations between operators and elements. For example, we assume that all elements are distinct, and that laws such as $a * b + a * c = a * (b + c)$ are not applicable. Under the above assumption, the algorithms presented generate optimal sequences of evaluation.

In practice, the associative and commutative laws need not and in most cases do not hold, as machines carry information of finite precision. It is conceivable that the order in which the operators and elements are specified leads to clever use of the available facilities. In such cases, reordering the computation may lead to overflows or underflows. However, most computations are not critically dependent on the order of evaluation—especially as compilers are fairly arbitrary in producing executable code. In these cases, an optimal sequence of evaluation can be arrived at by permuting the specified order of operations, without adversely affecting the accuracy of the result.

In showing that the sequence generated is optimal under the assumed conditions, we show that the algorithms minimize, independently, the number of "clear and

* Present address: Department of Electrical Engineering, Princeton University, Princeton, N. J.

adds" into registers and the number of "stores." The number of binary operations cannot be reduced, as the associative and commutative transformations do not change the number of operators in an expression. Later we show that minimizing the number of program steps also minimizes the number of storage references.

In Section 6 we briefly consider the timing requirements of the algorithms. We indicate, without proof, that the time taken is proportional to n, the number of nodes in the tree for the expression.

The algorithms are designed for a machine with unlimited storage and $N \geq 1$ general registers. The commands permitted are of the following kinds:

1. C(storage) → C(register)
2. C(register) → C(storage)
3. OP[C(register),C(storage)] → C(register)
4. OP[C(register),C(register)] → C(register)

Commands of types 1 and 2 transfer the contents of a storage location to the contents of a register, and vice versa. This makes them the equivalents of the CLear and Add (CLA) and STOre (STO) commands respectively. Type 3 commands perform an operation on two operands, the first in a register and the second in storage. It is important to note that commands of the type

$$\text{OP[C(storage),C(register)]} \rightarrow \text{C(register)}$$

are not permitted. This restriction is in keeping with the instruction sets of many present-day machines. For example, in a divide operation, the dividend is constrained to be in a register. Permitting commands of this type would have the same effect as making all operators commutative. Commands of type 4 apply an operator to the contents of two registers, leaving the result in a third (not necessarily distinct) register.

An *expression* is a sequence of binary operations on arguments. Arguments are either *initial* (defined externally) or *intermediate* (defined by operations on other arguments) values. Such expressions may be represented by binary trees. Initial values correspond to *leaves* and intermediate values to interior nodes of the tree. Every interior node represents a binary operation on the elements corresponding to its descendants. The value associated with a node can only be computed by applying the corresponding operator to the values of its two descendant subtrees taken in the proper order. Due to commands of type 3, the order in which the descendants are taken is important. Left descendants will be taken to correspond to the first operand and right descendants to the second operand in all commands. Two nodes in a tree are *twins* if they have the same ancestor.

A *program* is a sequence of operations which evaluate an expression when given the initial values in specified storage locations. The program terminates with the value of the entire expression in a register. The *cost* of a computation is the number of program steps required for the computation on the machine described.

Adopting an idea of Nakata [1], and Meyers [2],[1] we shall *label* each node with a number that turns out to be the minimal number of registers required to evaluate the node without stores. Given a tree, the algorithm defined later in this paper suitably labels it and uses these labels to define a minimal sequence of operations for the machine described.

In the first part of this paper, all operations are taken to be noncommutative. This restriction is later relaxed to allow associative and commutative operations.

[1] Our labels differ from these in order to handle noncommutativity correctly.

2. *The Basic Algorithm*

To each node η is assigned a *label* $L(\eta)$, from the bottom up. It should be noted that for the time being all operators are assumed noncommutative:

L1. If η is a leaf and the left descendant of its ancestor, then $L(\eta) = 1$; if η is the right descendant then $L(\eta) = 0$;

L2. If η has descendants with labels l_1 and l_2, then for $l_1 \neq l_2$, $L(\eta) = \max(l_1, l_2)$, and for $l_1 = l_2$, $L(\eta) = l_1 + 1$.

We now give an algorithm for the evaluation of arithmetic expressions using at most N registers. This algorithm is, in fact, the "obvious" top-down algorithm. Significantly, we prove its optimality under the assumptions previously mentioned.

ALGORITHM 1

After generating the tree and labeling the nodes:

(1) Apply (2) to the root with registers $b_1, b_2, \cdots, b_N$ available. Routine (2) will evaluate the expression represented by the subtree extending from the node to which it is applied. The result will appear in the lowest numbered register available.

(2) Let η be a node, $L(\eta) = l$ and $l > 0$. Suppose registers $b_m, b_{m+1}, \cdots, b_N$ are available, $1 \leq m \leq N$.

We first treat the case $l = 1$.

(A) If η is a leaf it must be a left descendant. Enter the value of η, which is an initial value, into b_m.

(B) If η is not a leaf its right descendant must be a leaf, else η's label would be at least 2. Apply (2) to its left descendant with registers $b_m, b_{m+1}, \cdots, b_N$ available. The result will appear in b_m. We then evaluate η in one step. The value of the left descendant is in b_m, the right in storage. Return the value in b_m.

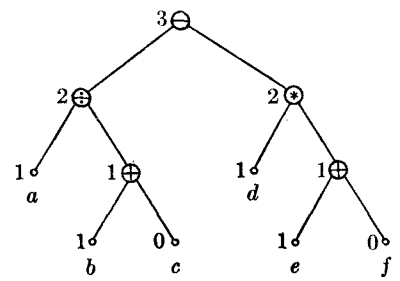
Now suppose $l > 1$. Let the descendants of η have labels l_1, l_2.

(C) If $l_1, l_2 \geq N$ apply (2) to the right descendant with registers $b_m, b_{m+1}, \cdots, b_N$ available. Store the value of the right descendant. Apply (2) to the left descendant with registers $b_m, b_{m+1}, \cdots, b_N$ available, leaving the result in b_m. Evaluate η using the value in b_m and the value of the right descendant in storage. Leave the result in b_m.

(D) If $l_1 \neq l_2$, and at least one of l_1, l_2 is less than N, apply (2) to the descendant with the higher label, with registers $b_m, b_{m+1}, \cdots, b_N$ available. Leave the result in b_m. Apply (2) to the other descendant with registers $b_{m+1}, b_{m+2}, \cdots, b_N$ available. The result will appear in b_{m+1}. Evaluate η using the values in b_m and b_{m+1}, and leave the result in b_m. Exception: If the smaller label is 0, do not apply (2), but rather evaluate η using the value in b_m and the value of η's right descendant, which must be a leaf, in storage.

(E) If $l_1 = l_2 < N$, treat the left descendant as the node with the higher label and proceed as above.

Example 1. Consider the expression $a/(b + c) - d*(e + f)$, whose tree is:



The integers at each node are the labels assigned by the labeling algorithm. Algorithm 1 produces the following code, if $N = 2$.

1. $d \rightarrow$ reg1
2. $e \rightarrow$ reg2
3. reg2 + $f \rightarrow$ reg2
4. reg1*reg2 $\rightarrow$ reg1
5. reg1 $\rightarrow T$
6. $a \rightarrow$ reg1
7. $b \rightarrow$ reg2
8. reg2 + $c \rightarrow$ reg2
9. reg1/reg2 $\rightarrow$ reg1
10. reg1 $- T \rightarrow$ reg1

In the above computation instructions 1, 2, 6, and 7 are CLA's. Instruction 5 is a store. The rest are binary arithmetic operations. It is possible to evaluate the given expression using only 8 instructions if $*$ is assumed commutative. Example 2 shows this.

Several papers have dealt with the compilation of code for arithmetic expressions and Algorithm 1 can be considered a generalization of some of those ideas. Floyd [4] suggested that the right argument of a noncommutative operation be computed first. Anderson [5] gives an algorithm which starts with an expression, constructs a binary tree, and then produces code for a machine with one register. Given such a machine, the code produced by the algorithm in this paper is essentially the same as the tree code in [5].

Nakata [1] and Meyers [2] have essentially similar results. As mentioned in Section 1, the algorithms in this paper are generalizations of these. In [2] a cost criterion is defined to be *distributive with respect to a given computer* if and only if each program for that computer which evaluates an expression in an optimal way (according to the criterion) also evaluates all subexpressions of that expression in optimal ways (according to the same criterion). The cost of computation defined in Section 1 is distributive with respect to our machine. When more than one register is involved, distributivity is not obvious, because different numbers of registers may be available to different nodes. The following sequence of lemmas, in effect, proves the distributivity.

We now show that Algorithm 1 produces a minimal cost program when all operators are noncommutative.

LEMMA 1. *Let Algorithm* 1 *be started with N registers available. Let r be the number of registers available when routine* (2) *of Algorithm* 1 *is applied to a node* η. *Then:* $r = N$ *if* $L(\eta) > N$; $N \geq r \geq L(\eta)$ *if* $L(\eta) \leq N$.

PROOF. We shall prove this lemma by induction on the number of applications of routine (2). The first time routine (2) is applied, it is for the root and all N registers are available.

Let the lemma hold for the first k calls, and let the $(k + 1)$-th call be for node η_1 with r registers available. Let η_1's twin node be η_2 and its ancestor η. Let their labels be l_1, l_2 and l respectively.

Case 1. $l_1 \geq N$. Since $l_1 \geq N$, we have $l \geq N$. As the lemma holds for the application of routine (2) to η, N registers were available then. From (2C), if $l_2 \geq N$, then when routine (2) is applied to η_1, N registers are available and the lemma holds. If $l_2 < N$, then N registers are available for η_1, by (2D).

Case 2. $l_1 < l_2$ and $l_1 < N$. Then from the labeling rules, $l_2 = l$ and $l_1 < l$. The

call for η_2 was before the $(k + 1)$-th and r registers were available, $r \geq l_2 = l$. By (2D), the $(k + 1)$-th call occurs with $r - 1$ registers. Since $l_1 < l \leq r$, we have $l_1 \leq r - 1$, and the lemma holds.

Case 3. $l_2 < l_1 < N$. Then $l_1 = l$. The call for η occurred before the $(k + 1)$-th call. Therefore $r \geq L(\eta) = l$. By (2B) or (2D), the $(k + 1)$-th call occurs with r registers and the lemma holds.

Case 4. $l_1 = l_2 < N$. In this case, from the labeling rules $l_1 = l_2 = l - 1$. Since routine (2) was called for η before it was called for η_1, $r \geq l$ registers were available. By (2E), routine (2) is called for η_1 with r or $r - 1$ registers depending on whether η_1 is the left or the right descendant of η respectively. Since $r \geq l$, we have $r - 1 \geq l - 1$, and the lemma holds.

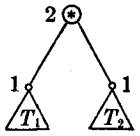
LEMMA 2. *Algorithm* 1 *evaluates a tree with no stores when as many registers as the label of the root are available.*

PROOF. Each interior node represents an operation. To evaluate a node, the algorithm evaluates both its descendants, then applies the operator. Since this is true for all interior nodes in the tree, Algorithm 1 correctly evaluates the expression. With Lemma 1, a simple induction on maximum path length of a tree shows that no stores occur.

LEMMA 3. *For trees with root labels up to* 2, *the label of the root is a lower bound on the minimum number of registers required to evaluate the tree without any stores.*

PROOF. Let l be the label of the root. Then from the labeling rules l is the highest label in the tree. The lemma is trivial for $l = 1$.

Consider a tree in which only the root has a label 2:



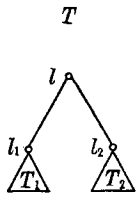
For this tree, at least two registers are required. This is because no matter how evaluation proceeds, even if T_1 is a leaf, its value must be in a register when the * operation is performed. Since T_2 is not a leaf, but the result of a computation, its value can only appear in a register, as no stores occur. Therefore at least two registers are required.

Any other case of a tree with a root labeled 2 must contain a subtree of the type just considered, proving the lemma.

LEMMA 4. *The label of a node is a lower bound on the minimum number of registers required to evaluate the node without any stores.*[2]

PROOF. From Lemma 3 this lemma holds for all labels up to and including 2.

Let T be a smallest tree for which a violation of the hypothesis occurs, then T can be represented by



[2] This result was proven in [3] using a similar but not identical labeling scheme and a different computer model. Our proof is more concise and carries over to the model of [3].

where the l's are labels. There are three possibilities for l_1 and l_2.

	l_1	l_2
1	l	$< l$
2	$< l$	l
3	$l - 1$	$l - 1$

For case 1(2) the lemma holds, otherwise the left (right) descendant will be a smaller violating tree.

For case 3, when the first command is executed, the value brought into a register is either for tree T_1 or for tree T_2. Since we have assumed that there are no nontrivial relations among operators and elements, the contents of a register at any time can be identified with a partial value in at most one of the subtrees, unless it is the value of the entire tree. If the initial register used is for tree T_1, there must be at least one register used for a partial value of tree T_1 until the final computation. Thus tree T_2 is evaluated using at most $l - 2$ registers, providing a contradiction. An analogous argument holds if the first register used is for tree T_2.

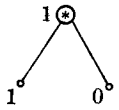
THEOREM 1. *If we assume no nontrivial relations between operators and elements, Algorithm* 1 *uses the minimum number of registers for the evaluation, with no stores, of an expression.*

PROOF. Immediate from Lemmas 2 and 4.

Definitions. Given a number of registers N, we say a node is *major* if both its descendants have labels at least as great as N. We say a node is *minor* if it is a leaf and the left descendant of its ancestor.

LEMMA 5. *The minimum number of stores used in the evaluation of an arithmetic expression with no nontrivial relations among operators and elements is equal to or greater than the number of major nodes in the tree for that expression.*

PROOF. We shall prove this lemma by induction on the number of nodes. For the smallest nontrivial tree,



the lemma is true as there are no major nodes, and no stores are required.

Suppose N registers are available. Let the lemma hold for trees of up to $k - 1$ nodes. The label of a major node must be at least $N + 1$. If the major node could be evaluated without any stores, then a contradiction of Lemma 4 would occur. Therefore, there must be at least one store in a program that evaluates a tree with a major node.

Consider a minimum store program P for a tree T with k nodes and $M \geq 1$ major nodes. When the first store occurs (and we have just shown that there must be one) the value of a subtree S of T is stored in a temporary. Replace S by a leaf to form a new tree T'. Revise P to evaluate T', by deleting those instructions which were a part of the computation of S and treating the temporary as the value of the new leaf. Call the new program P'.

Without loss of generality, we can assume that at least one operation was performed before the store occurred. That is, given any program that causes a store

before any operation is performed, there exists a program that performs the same computation with fewer stores. Thus, T' has fewer than k nodes, and the number of stores required for T' is equal to or greater than the number of its major nodes. P evidently has one store more than P'.

We will now show that the number of major nodes of T' is at least $M - 1$. In T, consider the first major node μ preceding S. (If there is none, we are done as no node of S can be major.) Let the descendant of μ that precedes S (if any) be μ_1. Otherwise, the root of S is μ_1. Let node μ_2 be the twin of μ_1. Since μ is major $L(\mu_1)$, $L(\mu_2) \geq N$. Let $L'(\cdot)$ give the labels in T'.

All nodes preceding S in T have their labels reduced by some amount in T'. It is true that μ may no longer be major, but μ_2 is not affected, so $L'(\mu_2) = L(\mu_2) \geq N$. Since $L'(\mu) \geq L'(\mu_2)$, we have $L'(\mu) \geq N$. Hence nodes preceding μ have labels equal to or greater than N in T'. Since μ precedes no major node, if M was the number of major nodes in T, there are at least $M - 1$ major nodes in T'.

By the inductive hypothesis, the number of stores required to evaluate T' is at least $M - 1$. Therefore, P' has at least $M - 1$ stores and P has at least M stores.

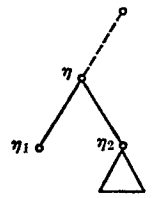
LEMMA 6. *If M is the number of major nodes in a tree, then the number of stores in its evaluation by Algorithm 1 is equal to M.*

PROOF. Routine (2C) is the only one of (2A)–(2E) which causes a store, and it causes exactly one store with each application. When (2) is applied to a node, routine (2C) is used if and only if the node is major.

LEMMA 7. *The number of CLA's in the evaluation of a tree is at least the number of minor nodes in the tree.*

PROOF. We shall prove this lemma by induction on the number of minor nodes. For trees with only one minor node, the lemma is easily seen to hold, as at least one CLA is necessary before evaluation can begin.

Let the lemma hold for trees with up to $k - 1$ minor nodes, and consider the following tree with k minor nodes:



η_1 is a leaf; $L(\eta_1) = 1$, and η_2 is the root of a subtree with $k - 1$ minor nodes.

To evaluate node η, the value of η_1 must be in a register as it is a left operand. (Note that we are still assuming that all operations are noncommutative.) Therefore, a CLA must occur for the value of η_1. Since the value of η_1 is not used in evaluating η_2, a program for the total tree which had fewer than k CLA's would yield a program to evaluate η_2 using fewer than $k - 1$ CLA's, contrary to hypothesis.

LEMMA 8. *Algorithm 1 evaluates a tree with as many CLA's as there are minor nodes in the tree.*

PROOF. Only routine (2A) is applied to leaves, and only to minor nodes at that. Each application causes a CLA.

THEOREM 2. *For expressions with noncommutative operators and no nontrivial relations between operators and elements, Algorithm 1 produces a minimal cost program for the machine we have described.*

PROOF. From Lemmas 5 and 6, the number of stores is minimal. From Lemmas 7 and 8, the number of CLA's is minimal. Each binary operation specified is performed once. Hence, the number of program steps is minimal.

3. *Commutative Operators*

We have now derived a property of trees, the sum of the number of major and minor nodes, which measures the speed with which its value can be computed. The results of Section 2 prove our measure to be correct. When certain algebraic transformations are permitted, we are not required to evaluate a given tree, but may evaluate any one of its equivalent trees, obtained by applying the allowed transformations. In this section and Section 4, we establish procedures for finding a best equivalent tree when the commutative and associative transformations are permitted. Other transformations may make the problem of finding best trees much harder than in the commutative and associative case.

When commutative operators are present, there is a family of trees that compute the same value. Members of this family are derived from each other by interchanging or "flipping" the descendants of commutative nodes. We will select that member of the family of trees that minimizes the highest label in the tree and the number of major and minor nodes.

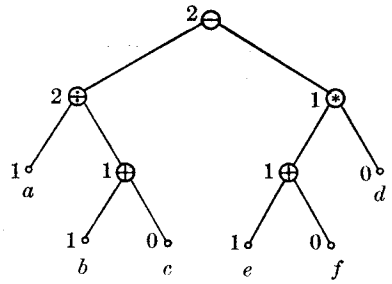
Looking back, it becomes evident that the only trees for which flipping will help are trees with minor nodes which correspond to commutative operators and have nonleaf right descendants. In such cases, flipping the leaf from the left to the right will reduce its label from 1 to 0, reduce the number of minor nodes, and possibly lead to a reduction in the highest label and/or number of major nodes.

Algorithm 1 may easily be modified to perform the flipping.

ALGORITHM 2

1. Generate labels.
2. For all nodes η such that $L(\eta) > 1$, its left descendant is a leaf, and the operator corresponding to η is commutative, flip the descendants so that the leaf is the right descendant (the restriction $L(\eta) > 1$ guards against flipping when both nodes are leaves).
3. Proceed as for Algorithm 1.

Example 2. If Algorithm 2 is used for the expression of Example 1, the expression actually evaluated is $a/(b + c) - (e + f)*d$, whose tree is:



The code for this tree is

1. $a \rightarrow$ reg1
2. $b \rightarrow$ reg2

3. reg2 + $c \rightarrow$ reg2
4. reg1/reg2 $\rightarrow$ reg1
5. $e \rightarrow$ reg2
6. reg2 + $f \rightarrow$ reg2
7. reg2*$d \rightarrow$ reg2
8. reg1 − reg2 $\rightarrow$ reg1

Definitions. Node η_1 in tree T_1 and node η_2 in tree T_2 are *corresponding* nodes if neither is a leaf, and the expressions represented by the subtrees with roots η_1 and η_2 may be obtained from each other by the commutative transformation. Let T_α represent a tree with root α, and let $\eta(T_\alpha, T_\beta)$ represent a tree with root η and left (right) descendant subtrees $T_\alpha(T_\beta)$. If γ is a leaf, we identify γ with T_γ.

LEMMA 9. *Of the family of trees that compute the same result as a given tree, the tree T generated by Algorithm 2 has the minimum number of major nodes, the minimum number of minor nodes, and the label of every nonleaf node in tree T is less than or equal to the label of the corresponding node of any other tree in the family.*

PROOF. We shall prove this lemma by induction on the maximum path length P in a tree. For a maximum path length of 1, the lemma is easily seen to hold.

Let it hold for $P \leq k - 1$, and consider a tree generated by Algorithm 2 with $P = k$ and a commutative operator at the root. The tree may be represented by $\eta(T_\alpha, T_\beta)$.

Case 1: β is a nonleaf. Then the other trees in the family may be of either of the following types—$\eta_1(T_{\alpha_1}, T_{\beta_1})$ or $\eta_2(T_{\alpha_2}, T_{\beta_2})$. Here α_1 and α_2 correspond to α, β_1 and β_2 to β, and η_1 to η_2 to η.

As the lemma holds for $P \leq k - 1$, we have $L(\alpha) \leq L(\alpha_1)$, $L(\alpha_2)$ and $L(\beta) \leq L(\beta_1)$, $L(\beta_2)$. Therefore, $L(\eta) \leq L(\eta_1)$, $L(\eta_2)$. If η is major then η_1 and η_2 are also major. Thus, the lemma holds in this case.

Case 2: β is a leaf. Then the other trees in the family are of either of the following types—$\eta_1(\beta_1, T_{\alpha_1})$ or $\eta_2(T_{\alpha_2}, \beta_2)$. The notation here is the same as for case 1. Since β is not a minor node, the number of minor nodes in the generated tree is minimal.

As β is a leaf, $L(\beta) = 0$. However, $L(\beta_1) = 1$ and $L(\beta_2) = 0$. Therefore $L(\beta) \leq L(\beta_1)$, $L(\beta_2)$. As the lemma holds for $P \leq k - 1$, we have $L(\alpha) \leq L(\alpha_1)$, $L(\alpha_2)$. Therefore, just as for case 1, the lemma holds.

THEOREM 3. *Algorithm 2 produces an optimal sequence of evaluation for an expression, on the assumption that there are only commutative and noncommutative operators in the expression and that there are no other nontrivial relations between operators and elements in the expression.*

PROOF. The operations performed are STO's, CLA's, and binary operations. By Lemmas 5–9 the number of STO's and CLA's is minimal. Each binary operation specified is performed once. Hence the number of operations is minimal and the sequence of evaluation is optimal.

4. *Associative Operations*

Exploiting the associative and commutative properties of some operators, we achieve reductions in the register requirements and the time of a computation. The example shown in Figure 1, in which both associativity and commutativity hold, illustrates these reductions.

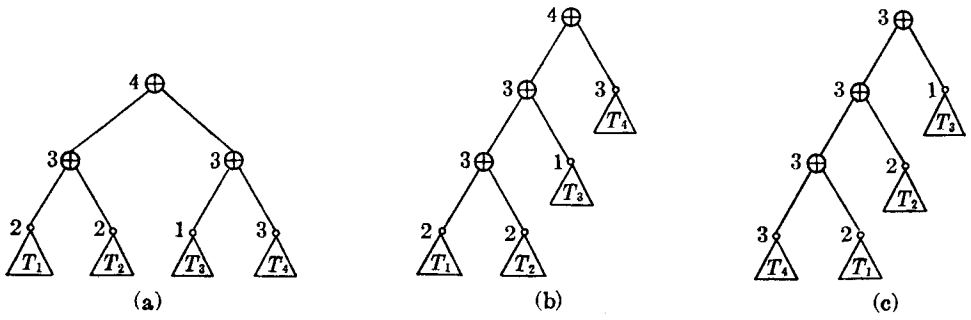


FIG. 1

The algorithm presented in this section would produce the tree in Figure 1(c) when started out with either of the other trees in Figure 1.

It should be noted that the only associative operators we consider are those that are both associative and commutative (a-c). In practice, most operators that are associative are also commutative on integers, reals, etc.[3]

Definitions. An operator is an a-c *operator* if it is both associative and commutative. A *cluster* is a set of nodes in a tree, such that

(1) each node in the set is associated with the same operator,

(2) the branches between nodes in the set form a tree, and

(3) if the operator associated with the set of nodes is not an a-c operator the set contains only one element.

The *root of a cluster* is the node in the cluster that precedes all other nodes in the cluster. The *ancestor of a cluster* is the ancestor of the root of the cluster. The *descendants of a cluster* are the descendants of the leaves in the cluster. The *descendant subtrees of a cluster* are the subtrees preceded by the descendants of the cluster. A *maximal cluster* is a cluster which is not a proper subset of any other cluster.

Given a binary tree, identify the maximal clusters. It should be clear that these can be found uniquely. Let a maximal cluster C have descendants $\delta_1, \delta_2, \cdots, \delta_m$ and let the root of C be a descendant of maximal cluster A. $\delta_1, \delta_2, \cdots, \delta_m$ are roots of clusters $D_1, D_2, \cdots, D_m$. Generate a tree with node μ in place of cluster C. Let μ have descendants $\delta_1, \delta_2, \cdots, \delta_m$. Proceed from the top down generating a tree with one node per maximal cluster. In the new tree, there is a single node for each maximal cluster in the binary tree. The new tree is called an *associative tree.*

The above definitions have the following motivation. Let T be a binary tree and A its corresponding associative tree. Let T' be derived from T by a single associative or commutative transformation. Then the associative tree A' corresponding to T' can be obtained from A by permuting the descendants of one node of A. That node corresponds to the maximal cluster of T in which the transformation took place. Conversely, any permutation of descendants of a node of A can be mirrored by the application of a sequence of a-c transformations on one cluster of T.

Let F be the family of trees derivable from T by a-c transformations. The problem of finding the member of F yielding the best program by Algorithm 1 is thus broken into two parts. First, using permutations of descendants of the associative nodes, we find the "best" ordering for descendants. Then, from the resulting associative

[3] D. M. Ritchie pointed out that the FORTRAN operation SIGN(a, b) is associative but not commutative.

tree, we construct that one of the corresponding binary trees which yields the minimal cost program.

The labeling rules for associative trees are:

LA1. If η is a leaf and the leftmost descendant of its ancestor, then $L(\eta) = 1$. For all other leaves, $L(\eta) = 0$.

LA2. If η has descendants with labels $l_1, l_2, \cdots, l_m$, $l_1 \geq l_2 \geq \cdots \geq l_m$ then $L(\eta) = l_1$ if $l_1 > l_2$, $L(\eta) = l_1 + 1$ if $l_1 = l_2$.

ALGORITHM 3

1. Generate a binary tree for the expression. Convert the binary tree to an associative tree.
2. Label the nodes according to the labeling rules for associative trees, and do steps 3 and 4 from the bottom up.
3. At every associative node η, order the descendants by label. Let the new order be $\delta_0, \delta_1, \cdots, \delta_m$ such that $L(\delta_0) \geq L(\delta_1) \geq \cdots \geq L(\delta_m)$. If possible, let δ_0 not be a leaf. Replace the associative node η by m nodes η_i, $1 \leq i \leq m$, such that η_{i+1} is the ancestor of η_i, $1 \leq i < m$. The left descendant of η_{i+1} is η_i and the right descendant is δ_i, $1 \leq i \leq m$. The left and right descendants of η_1 are δ_0 and δ_1 respectively. The ancestor of η is now the ancestor of η_m.
4. If the node η is not associative, proceed as for Algorithm 2.

LEMMA 10. *Let N be the number of available registers. Given a binary tree containing a cluster with m descendant subtrees, r of which have roots with labels greater than or equal to N, the cluster contains at least $r - 1$ major nodes.*

PROOF. We shall prove this lemma by induction on the maximum path length P of the subtree dominated by the cluster (i.e. the tree formed by considering the cluster and its descendant subtrees only).

For $P = 1$, the lemma holds trivially. Let the lemma hold for $P < k$, and consider a tree with $P = k$, say $\eta(T_{\eta_1}, T_{\eta_2})$.

If η_1 and η_2 are not in the cluster for η the lemma holds trivially. If only one of η_1 and η_2 is in the cluster, the induction is elementary.

When both η_1 and η_2 are in the cluster there are three possible cases. Let the nodes of the cluster in subtrees T_{η_1} and T_{η_2} have r_1 and r_2 descendant subtrees whose roots have labels greater than or equal to N. The three cases are:

1. $r_1 = r_2 = 0$;
2. one of r_1, r_2 is 0;
3. $r_1, r_2 > 0$.

Case 1. Elementary.

Case 2. Without loss of generality, let $r_2 = 0$ and $r_1 = r$. By the inductive hypothesis, the portion of the cluster in subtree T_{η_1} has $r - 1$ major nodes.

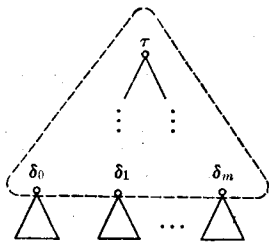
Case 3. The lemma holds for the portions of the cluster dominated by η_1 and η_2. Hence they have $r_1 - 1$ and $r_2 - 1$ major nodes respectively. Since $r_1, r_2 > 0$, $L(\eta_1), L(\eta_2) \geq N$. Hence η is major, and the number of major nodes in the tree is $r_1 + r_2 - 1$. As $r_1 + r_2 = r$, the lemma holds.

LEMMA 11. *Of the family F of binary trees that compute the same result as a given tree, Algorithm 3 generates that tree T, that has the minimum register requirements for evaluation without stores and the minimum number of major nodes of all trees in F.*

PROOF. We shall prove this lemma by induction on the maximum path length P in the tree T generated by Algorithm 3.

For $P = 0$ or 1 the lemma holds trivially.

Let the theorem hold for $P < k$, and consider a tree with $p = k$:



Consider the maximal cluster C defined by the root τ of the tree T. The cluster C has $m + 1$ descendants $\delta_0, \delta_1, \cdots, \delta_m$. The lemma holds for each descendant subtree of the cluster so $L(\delta_i)$, $0 \leq i \leq m$ are minimal and the number of major nodes in each subtree is minimal.

From Lemma 1, if r of the $L(\delta_i)$, $0 \leq i \leq m$ are greater than or equal to N there are at least $r - 1$ major nodes in the cluster C. The reader can show that in Algorithm 3, the cluster produced from C has only $r - 1$ major nodes. The number of major nodes in tree T is therefore minimal.

As for the register requirements, from Algorithm 3, we can assume $L(\delta_i) \geq L(\delta_{i+1})$, $0 \leq i \leq m$.

Case 1. $L(\delta_0) > L(\delta_1)$. Since τ precedes δ_0, $L(\tau) \geq L(\delta_0)$. As in tree T, $L(\tau) = L(\delta_0)$, the label of τ is minimal.

Case 2. $L(\delta_0) = L(\delta_1)$. As $L(\delta_0) = L(\delta_1)$, a node preceding both δ_0 and δ_1 must have a label at least one greater than $L(\delta_0)$. In tree T, $L(\tau) = L(\delta_0) + 1$, so the label of τ is minimal. Any other tree in the family F with the same descendant subtrees must have at least as many major nodes and require at least as many registers as T. From the inductive hypothesis, the same is true of the descendant subtrees themselves. Hence the lemma holds.

THEOREM 4. *Algorithm 3 produces an optimal sequence of evaluation for an expression on the assumption that there are only commutative, noncommutative, and a-c operators in the expression and that there are no other nontrivial relations between operators and elements in the expression.*

PROOF. From Lemma 11, the number of STO's is minimized. As Algorithm 2 is applied to the binary tree generated, and hence to the nodes with commutative but not a-c operators, no descendant of a cluster is made a minor node unless all descendants are leaves. The number of CLA's is therefore minimized by Theorem 3.

The number of binary operations performed is equal to the number specified in the expression. (Note that the associative and commutative transformations do not change this number.) Hence, the sequence of evaluation is optimal.

5. *Storage References*

Every time a piece of information is obtained from or placed in storage, a "storage reference" occurs. The information referred to may be an instruction, an initial value, or a piece of temporary storage.

The cost structure we have used so far is not based on the number of storage references during evaluation. However, it is shown in this section that Algorithms 1, 2, and 3 minimize the number of storage references as well as the number of operations.

LEMMA 12. *Let E be an expression whose tree T has n nodes. Let c be the minimum number of CLA instructions in any program evaluating E and s the minimum number of STO's in any program evaluating E. Then the number of storage references in any program evaluating E is at least $c + n + 3s$.*

PROOF. Let n_0 be the number of nonleaves in T and n_l the number of leaves; $n_0 + n_l = n$. Any program for E has at least $c + n_0 + s$ steps, since n_0 is the number of binary operations to be performed. At least one reference to each initial value in storage must occur, else there would be an identity of the form $E' = E$, where E' was missing at least one of the variables of E. There are n_l initial values. Each time a partial value is stored, a storage reference occurs, and at least s of the values stored are referenced later. (For a value not subsequently referenced need not be stored, and s is a presumed lower bound on the number of storages.) Thus there are at least $n_l + 2s$ storage references involving data. The minimum number of storage references is thus $(c + n_0 + s) + (n_l + 2s) = c + n + 3s$.

THEOREM 5. *Algorithms* 1, 2, *and* 3 *minimize the number of storage references in the evaluation of a given expression.*

PROOF. The programs produced by these algorithms have both the minimum number of CLA's and the minimum number of STO's under their respective assumptions. It is easy to verify that the number of storage references is equal to the lower bound of Lemma 12.

6. *Discussion*

For any algorithm to be of practical utility, it must evaluate its object in a reasonable number of steps. The adjective reasonable is not precisely defined, and we will not attempt to do so. We merely state that once the tree has been set up, the algorithms presented in this paper require a number of steps linearly proportional to the number of nodes in the tree.

For Algorithms 1 and 2, the first time each node is visited is when the labeling is done. The second and third visits occur when we travel down from the root to the leaves to determine the order of evaluation, and backtrack, producing the evaluation sequence as we go.

For Algorithm 3, two extra passes occur when the associative tree is constructed, then remade into a binary tree. Further overhead is introduced by the need to reorder the descendants of the associative nodes. A careful look at the algorithm reveals that it is not necessary to order all the descendants by label. All we need to find is the descendant with the maximum label if it is unique, and two descendants with maximal labels if there is no unique maximum label. This operation can be done in time proportional to the number of nodes involved. The associative node can then be split into binary nodes in linear time. Thus, a slight modification of Algorithm 3 can be done in linear time.

It is easy to modify the algorithms to take into account operations that call for extra registers. An example of such an operation is function invocations which need extra registers for linkages. Labeling rule L2 of Section 2 has then to be modified.

It can be shown that rule L2 at node η with descendants α and β is equivalent to $L(\eta) = \min\{\max[L(\alpha), L(\beta) + 1], \max[L(\beta), L(\alpha) + 1]\}$.

If the operation at node η requires R registers then rule L2 may be replaced by $L(\eta) = \max[R, \min\{\max[L(\alpha), L(\beta) + 1], \max[L(\beta), L(\alpha) + 1]\}]$.

ACKNOWLEDGMENTS. The authors are indebted to A. V. Aho, R. H. Canaday, L. A. Dimino, B. W. Kernighan, M. D. McIlroy, and D. M. Ritchie for helpful discussions and comments.

REFERENCES

1. NAKATA, IKUO. On compiling algorithms for arithmetic expressions. *Comm. ACM 10*, 8 (Aug. 1967), 492–494.
2. MEYERS, W. J. Optimization of computer code. Unpublished memorandum, G. E. Research Center, Schenectady, N. Y., 1965 (12 pp.).
3. REDZIEJOWSKI, R. R. On arithmetic expressions and trees. *Comm. ACM 12*, 2 (Feb. 1969), 81–84.
4. FLOYD, R. W. An algorithm for coding efficient arithmetic operations. *Comm. ACM 4*, 1 (Jan. 1961), 42–51.
5. ANDERSON, J. P. A note on some compiling algorithms. *Comm. ACM 7*, 3 (March 1964), 149–150.

RECEIVED OCTOBER, 1969; REVISED APRIL, 1970