1970

On the Number of Registers Needed to Evaluate Arithmetic Expressions

Victor Schneider

Report Number:
70-047

ON THE NUMBER OF REGISTERS NEEDED TO EVALUATE ARITHMETIC EXPRESSIONS

Victor Schneider

September 4, 1970
Revised
CSD TR47

Computer Sciences Dept.
Purdue University
Lafayette, Indiana 47907

Supported in part by NSF Grant GJ-851.

ABSTRACT

The question of how many temporary storage registers are needed to evaluate compiled arithmetic and masking expressions is discussed. It is assumed that any combination of left-to-right, right-to-left, top-to-bottom, and bottom-to-top techniques may be used to evaluate an expression, but that no factoring or re-arranging of the expression may occur. On this basis, the maximum number of registers needed to evaluate non-parenthesized expressions is N+1, with N the number of dyadic operator precedence levels. For parenthesized expressions with a maximum of K nested parenthetical subexpressions, the maximum number of registers needed is (K+1)N+1.

Key Words and Phrases: arithmetic expression, compiler design, dependency tree, programming, storage minimization.

C R Categories: 4.12, 4.22, 6.32

This paper looks at the question of how many temporary storage registers are called for in the design of a computer for performing combinations of arithmetic and masking operations. It assumes that machine-language code is to be compiled from expressions in some FORTRAN or ALGOL-like language, and that the compiler can evaluate its expression tree in any sequence whatever, but without transforming that tree. The target-language computer is assumed to have two or more operand registers, and we will represent statements in its machine language using the following notation:

```
Acc1   ←   Acc2 + Acc3
Acc2   ←   name1
name1  ←   Acc3
```

In these statements, name1 refers to a location in core storage, all operations must be performed on operands stored in the numbered registers, and some mechanism exists for storing data from core into these registers and for removing data from registers to core.

In what follows, we will use as an example an ALGOL-like language of arithmetic expressions having two operator precedence levels. The operations will be represented by the operators "+" and "×" having the conventional interpretations and priorities. We will first discuss arithmetic expressions that are written entirely without parentheses, and deduce the number of accumulators that are needed to evaluate the translations of such expressions. We can then extend our results to the case of expressions having a larger number of operators and precedence levels.

To illustrate the process of code generation from programming language expressions, we will represent the expressions in terms of dependency tree diagrams such as those suggested by J. Cohen (1) and

D. T. Ross (4). For example, the dependency diagram of Figure 1 represents the expression $a + b \times c + d$:

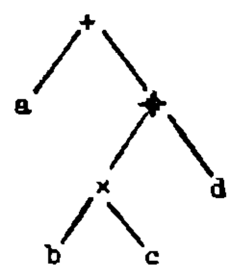


Figure 1. The Dependency Tree for $a + b \times c + d$

It should be noted that the dependency tree diagram for a general expression can be constructed using rather well-known compiler techniques (1,5). On an intuitive basis, we can say that these trees are constructed so that each operator is the root of a binary tree whose subtrees are the operands. Hence, because higher precedence operations are more "binding" than lower precedence operations, we can deduce the following tree diagram property:

> <u>In the dependency tree diagram of a parentheses-free expression, operators of lower precedence dominate (i.e., take as operands) operations of the same or higher precedence.</u>

Thus, in our diagram, the plus signs appear above the multiplication symbol. The dependency tree in which a multiplication symbol dominates operations of addition can only represent a parenthesized expression in which parentheses are used to delimit subexpressions that are treated like single operands.

In terms of a single-scan, left-to-right compiling algorithm such as that given in the appendix, the following code would be generated for the expression in Figure 1:

```
Acc1   ←   b
Acc2   ←   c
Acc1   ←   Acc1 × Acc2
Acc2   ←   d
Acc1   ←   Acc1 + Acc2
Acc2   ←   a
Acc1   ←   Acc2 + Acc1  .
```

As can be seen in the example above, the compiling algorithm proceeds in steps to generate code for evaluating the leftmost subtree in which the operator root dominates two variables. In each step, code is written for this leftmost subtree, after which this subtree is replaced in the tree by an accumulator name (denoting an intermediate result), which is treated as a variable in the following step. Thus, the first code for our example was the subtree (b × c), followed by ((b × c) + d), etc.

We can deduce from this example the following interesting property of non-parenthesized expressions:

Property 1: <u>Any non-parenthesized expression in which the operators are written in descending (or ascending) order of precedence requires at most two accumulators to execute the translated code.</u>

This is true because, in the dependency trees for such expressions, each operation dominates either two variables or one variable and one operation. Thus, for an expression written in descending (alternatively, in ascending) order of operator precedence, the leftmost subtree (rightmost subtree) has the highest precedence operator as root dominating two variables. The remainder of the tree has operators dominating operations as left subtrees (right subtrees) and variables as right subtrees (left subtrees).

Hence, when the code is compiled for this class of expression trees, each step of the code generation process causes a subexpression to be combined with a variable, leaving a subexpression result in a single operand register. Any unary operations only affect the intermediate result already stored in the single operand register. By induction, this result can be extended to languages of expressions having any finite number of binary operator precedence levels.

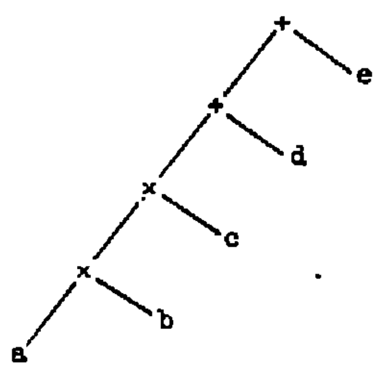


Figure 2. The Dependency Tree for $a \times b \times c + d + e$

Code Produced from the Dependency Tree:

Acc1	←	a	Acc2	←	d
Acc2	←	b	Acc1	←	Acc1 + Acc2
Acc1	←	Acc1 × Acc2	Acc2	←	e
Acc2	←	c	Acc1	←	Acc1 + Acc2
Acc1	←	Acc1 × Acc2			

As is apparent from Figure 1, we can generalize Property 1 as follows:

Property 2: Any non-parenthesized expression in which the operators are written in ascending followed by descending order requires at most two accumulators to execute the translated code.

This is so because, in the resulting dependency tree, the leftmost subtree in which the operator dominates two variables is at the bottom of a left-branching, descending order subtree whose root is the rightmost node of an ascending order subtree. Thus, this left-branching subtree will be translated into code using at most two accumulators, exactly as in Property 1. When this first left-branching subtree is translated, the compiler is left with a right-branching tree whose rightmost operator dominates a variable on the left, and the intermediate result denoting the first subtree on the right. The compiler then proceeds to evaluate this right-branching subtree, exactly as in Property 1.

As we see next, it is not true that non-parenthesized expressions written in descending followed by ascending order can be evaluated using two registers. Given our knowledge of how non-parenthesized expressions are evaluated it is interesting to determine what maximum number of operand registers are needed to execute translated code for more general expressions. Consider our left-to-right, single-scan code generation algorithm. In this algorithm, when an operator is the root of a tree having operators as roots of each subtree, the left subtree will be translated first, and its value will be stored in a temporary register while the right subtree is evaluated. If the right subtree in turn is of the same form, there results a second intermediate value stored in an operand register while a right subtree is evaluated, and so on. The question then becomes, "For how many levels can we force this process of storing intermediate results to continue?" The answer is that the maximum number of levels is the same as the maximum number of binary operator precedence levels in the expression language. We can see this property most clearly in the example given in Figure 3, where a third level of operator precedence, symbolized by division "÷" is included for clarity:

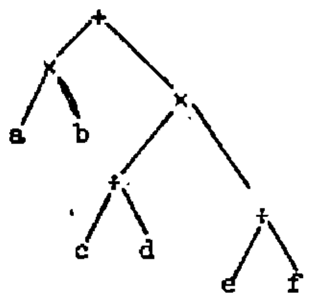


Figure 3. Dependency Tree for a × b + c ÷ d × e ÷ f

In this right-branching tree, each operator dominates either two variables or two operations of higher precedence. Because of this pyramid of operations, the single-scan compiler causes the intermediate result for (a × b) to be stored in one register, the intermediate result for (c ÷ d) is stored in a second register, and then (e ÷ f) is evaluated using two more temporary registers.

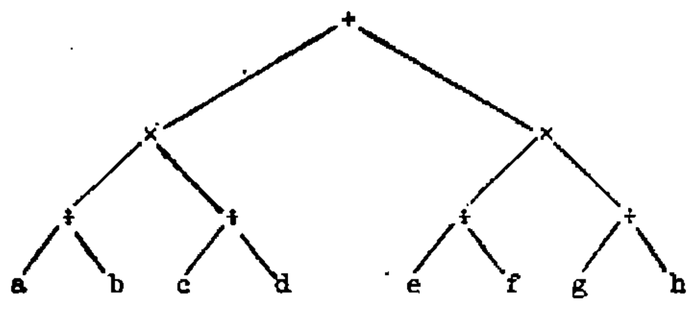
Translated Code Produced by a Left-to-Right Compiler for the Expression in Figure 3.

```
First Subtree:     Acc1   ←   a
                   Acc2   ←   b
                   Acc1   ←   Acc1 × Acc2
```

Second Subtree:	Acc2	←	c
	Acc3	←	d
	Acc2	←	Acc2 ÷ Acc3
Third Subtree:	Acc3	←	e
	Acc4	←	f
	Acc3	←	Acc3 ÷ Acc4
Result 1:	Acc2	←	Acc2 × Acc3
Final Result:	Acc1	←	Acc1 + Acc2

Thus, at most four operand registers are used to evaluate the compiled three-level, non-parenthesized expression.

As is obvious from inspection of Figure 3, a compiler that is not confined to single-scan, left-to-right translation could easily generate three register code for our expression. (For example, a right-to-left translator could produce three-register code.) We could then inquire into the possibility of rewriting the expression in Figure 3 so that <u>any</u> compiling scheme would call for the use of at least four registers. By exploiting symmetry, we can arrive at the expression illustrated in Figure 4 for which any compiling algorithm must use four temporary stores:



<u>Figure 4. A Non-Optimizable Parentheses-Free Expression,</u>
<u>a ÷ b × c ÷ d + e ÷ f × g ÷ h</u>

Because of the symmetry of the dependency tree in Figure 4, any compiling algorithm must work upward from the bottommost level of operations, and must therefore store temporary results of the first subtrees evaluated while the remaining subtrees are being evaluated. Suppose that we next attempt to alter the expression of Figure 4 by increasing its length. It soom becomes apparent that altering the left or right subtree of Figure 4 can at most cause Acc4 to be used more frequently during execution. Thus, in the worst cases for parentheses-free expressions of three precedence levels, at most four accumulators are needed to execute the translated code.

We can next attempt to extend these results inductively to the case of parentheses-free expressions with more precedence levels. By using the operations pyramid diagrammed in Figure 4, it is easy to show that a one-level language uses two operand registers, and a two-level language uses three. Thus, the hypothesis is true for up to three operator levels. If we take a fictitious language of expressions in which the operators are >, +, ×, and division in order of precedence, we can construct an operations pyramid, in which the ">" symbol dominates two copies of the tree in Figure 4 whose execution in assembly language uses five accumulators. Moreover, by observing that unary operations of any precedence do not increase the number of accumulators needed for execution, we have inductively demonstrated that

Property 3: N+1 accumulators are needed to execute the compiled code produced for a parentheses-free expression language having N precedence levels of binary operators.

Finally, we can consider the most general case of a parenthesized expression, and ask how such an expression can be written so as to force the use of a maximum number of registers. Consider first the case of a left-to-right, single-scan compiler. If we look at the worst-case dependency tree in Figure 3, we see that each variable in the tree can be replaced by a parenthesized expression. However, the maximum use of registers only arises when the rightmost operator (in this diagram, the "÷" symbol) dominates two parenthesized subexpressions. This is because each preceding subtree claims one operand register to store temporary results, and therefore, the maximum storage of temporary results occurs when the rightmost operation in turn dominates a further right-branching tree, as shown in Figure 5:

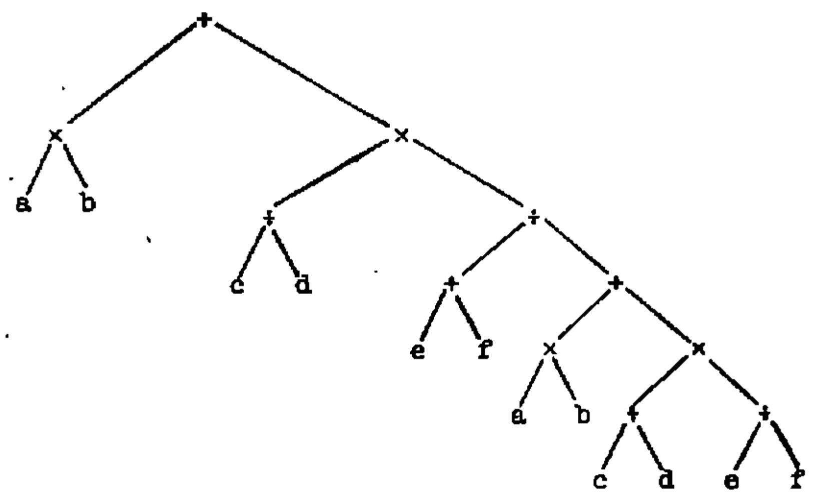


Figure 5. Dependency Tree for $a \times b + c \div d \times (e + f) \div (a \times b + c \div d \times e \div f)$

Note in Figure 5 that the choice of $(e + f)$ as the left operand in the right subtree under "÷" is arbitrary. Any other subexpression that forces the single-scan compiler to produce code and generate a temporary result in an operand register would do as well. As we can see from the branching pattern of this tree, five registers will be in use for storing temporary results while the rightmost subtree $(e \div f)$ is being evaluated. Hence, a maximum of seven registers will be in use at any one time during evaluation of this expression. If we replace the rightmost $(e \div f)$ in Figure 5 by $(e + f) \div (a \times b + c \div d \times e \div f)$ we see immediately that the maximum number of registers in use is now ten. Continuing this way, we arrive at the following result.

Property 4: <u>$(K+1)N+1$ operand registers are needed to execute the code produced by a non-optimized compiler for general parenthesized expressions having N levels of binary operators.</u>

The surprising fact is that this result also holds for so-called "optimized" compilers; in fact, it will be shown to hold for any scheme of code generation whatever. Our expedient is simply to make Figure 5 symmetrical in the same way that Figure 3 was made symmetrical. With this symmetry, it will then be apparent that, no matter which subtree is chosen first in the code-generation process, a total of $(K+1)N+1$ operand registers will be called into use upon execution. This symmetrical dependency tree is shown in Figure 6.

Acknowledgments

The author expresses his gratitude to Professor M. H. Halstead of Purdue University for many helpful suggestions. Research for this paper was supported in part by N.S.F. grant GJ-851.

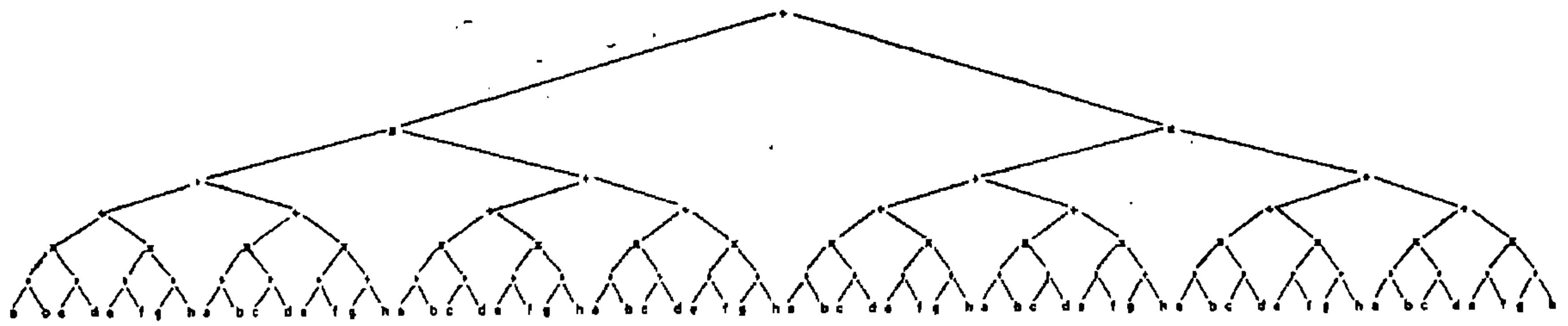

a bc de fg ha bc de fg ha bc de fg ha bc de fg ha bc de fg ha bc de fg ha bc de fg ha bc de fg h

Figure 6. A Symmetrical Dependency Tree Based on Figures 4 and 5:

÷(a ÷ b × c ÷ d + e ÷ f × g ÷ h) ÷ (a ÷ b × c ÷ d + e ÷ f × g ÷ h)
×(a ÷ b × c ÷ d + e ÷ f × g ÷ h) ÷ (a ÷ b × c ÷ d + e ÷ f × g ÷ h)
+(a ÷ b × c ÷ d + e ÷ f × g ÷ h) ÷ (a ÷ b × c ÷ d + e ÷ f × g ÷ h)
×(a ÷ b × c ÷ d + e ÷ f × g ÷ h) ÷ (a ÷ b × c ÷ d + e ÷ f × g ÷ h)

APPENDIX

A Translation Grammar for a Two-Level Language of Arithmetic Expressions (Notation from Wirth & Weber (6))

Syntactic Rules:

P_1: <var> → <name>

P_2: <factor> → <var>

P_3: <factor> → (<sum>)

P_4: <term> → <term> × <factor>

P_5: <term> → <factor>

P_6: <sum> → <sum> + <term>

P_7: <sum> → <term>

P_8: <stat> → <var>: = <sum>

Translation Rules:

Ψ_1: k ← k + 1 ; O[k] ← hash(name); type [k] ← 'reference';

Ψ_2, Ψ_3, Ψ_5: empty

```
Ψ4: if type [k-1] = 'reference' then (ℓ ← ℓ+1; p ← p+1;
       output [ℓ] ← 'Acc{p} ← {O[k-1]}';
       type [k-1] ← 'value'; O[k-1] ← p) else
    if type [k] = 'reference' then (ℓ ← ℓ+1; p ← p+1;
       output [ℓ] ← 'Acc{p} ← {O[k]}';
       type [k] ← 'value'; O[k] ← p) else undefined;
       p ← min (O[k-1], O[k]); ℓ ← ℓ+1;
       output [ℓ] ← 'Acc{p} ← Acc{O[k-1]} × Acc{O[k]}';
           k ← k-1; O[k] ← p;
```

Ψ_6: Similar to Ψ_4.

Ψ_7: empty

```
$\Psi_8$:  if type [k] = 'reference' then
             (p ← p+1; ℓ ← ℓ+1;
             output [ℓ] ← 'Acc{p} ← {O[k]}';
             type [k] ← 'value'; O[k] ← p) else undefined;
         p ← p+1; ℓ ← ℓ+1;
             output [ℓ] ← '{O[k-1]} ← Acc{O[k]}';
             O[k-1] ← O[k]; type [k-1] ← type [k]; k ← k-1;
```

BIBLIOGRAPHY

1. Cohen, J. Langages pour l'écriture de compilateurs. Doctoral dissertation, University of Grenoble, Grenoble, France, 1967.

2. Nakata, I. On compiling algorithms for arithmetic expressions. Comm. ACM 10, 8 (August 1967), 492-494.

3. Redziejowski, R. R. On arithmetic expressions and trees. Comm. ACM 12, 2 (February 1969), 81-84.

4. Ross, D. T. The AED approach to generalized computer-aided design. Proc. 22nd National ACM Conf., 1967.

5. Schneider, V. B. A system for designing fast programming language translators. Proc. 1969 Spring Joint Comp. Conf.

6. Wirth, N. and Weber, H. A generalization of ALGOL and its formal definition: Parts I and II. Comm. ACM 9, 1 (Jan. - Feb. 1966), 89-99.