One-Pass Compilation of Arithmetic Expressions for a Parallel Processor

HAROLD S. STONE
Stanford Research Institute, Menlo Park, California

Under the assumption that a processor may have a multiplicity of arithmetic units, a compiler for such a processor should produce object code to take advantage of possible parallelism of operation. Most of the presently known compilation techniques are inadequate for such a processor because they produce expression structures that must be evaluated serially. A technique is presented here for compiling arithmetic expressions into structures that can be evaluated with a high degree of parallelism. The algorithm is a variant of the so-called "top-down" analysis technique, and requires only one pass of the input text.

1. Introduction

The rapid advances in computer technology in recent years have brought us into an era in which speed of computation is limited by factors other than the operating speeds of its component devices. Two such factors, for example, are the input-output information bandwidths, and the propagation delays between the component devices of the system. In order to increase computation rate in view of these factors, the trend has been toward computer systems capable of a high degree of parallelism.

One approach to this end (by no means the only one) is exemplified by the central processor of the CDC 6600 which has several independent arithmetic units that are capable of fully parallel operation. If this approach is carried further in the coming years, processors may contain tens or hundreds of independent arithmetic units. The problem of efficient utilization of such a processor then becomes a major problem.

Ultimately, the problem of efficiency will fall on the compilers and the compiler writers. To this end we describe a one-pass algorithm for the compilation of expressions such that the resulting expression structure is inherently parallel. (By this we mean that elements of structure can be evaluated in parallel.) Most of the commonly used compilation techniques result in structures that must be evaluated serially. In Section 2 of this paper we describe the structures produced by several compilation techniques and relate these structures to the process of parallel computation. Section 3 contains a description of the expression compiler, and the ALGOL text of the compiler appears in an Appendix.

The research reported in this paper was sponsored by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, Bedford, Mass., under Contracts AF 19(628)-5920 and AF 19(628)-2902.

2. Expression Structures

The technique for translating conventional notation of arithmetic expressions into a "suffix Polish" string is well known (see, for example, [1]), but the resulting instruction sequence is not well suited to a parallel processor. In particular, the expression

$$A + B + C + D + E + F + G + H$$

is translated into the structure shown in Figure 1(a), which calls for seven additions to be performed sequentially. The translation algorithm essentially breaks the expression into partial sums as if the expression were parenthesized as

$$((((((A + B) + C) + D) + E) + F) + G) + H.$$

For a parallel processor of the type described in the Introduction, the optimum sequence of partial sums would correspond to a binary tree as indicated in Figure 1(b) and would be equivalent to

$$((A + B) + (C + D)) + ((E + F) + (G + H)).$$

The translation algorithm described in this paper gives the desired result. It always produces a full binary tree of n levels when there are 2^n terms, and unbalanced partial trees when the number of terms is not a power of two. Although this scan technique is claimed to be novel there has been flowcharted in the literature a one-pass scan that produces a Polish string in which terms are combined pairwise, and the resulting partial sums added sequentially [2]. For the example above, this scan would yield the third tree shown in Figure 1(c), which is equivalent to the expression

$$(((A + B) + (C + D)) + (E + F)) + (G + H)).$$

Hellerman [3] and Squire [4] have both reported multipass algorithms for compiling arithmetic expressions. Although these algorithms are set in a slightly different context, both produce results equivalent to the algorithm described here.

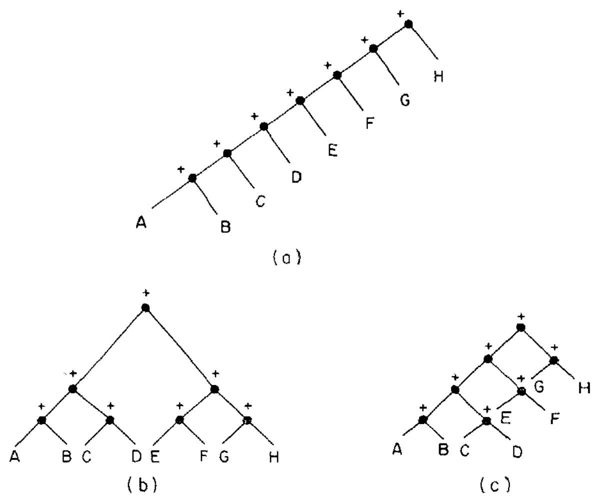

FIG. 1. Three different structures for the expression
$A + B + C + D + E + F + G + H$

It is worthwhile to note that none of the proposed translation techniques for parallel evaluation of expressions can comply with the rules for evaluating expressions in Algol [6] but all satisfy the less restrictive rules of ASA Fortran [7].

Before proceeding to a description of the algorithm, some discussion about the form of the translator output is pertinent to the material. It is normal practice to write suffix Polish in the form $AB + CD ++ EF + GH +++$ for the full binary tree, and $AB + CD ++ EF ++ GH ++$ for the suboptimal form of Figure 1(c). In this notation the symbol "A" corresponds to the operation "*Fetch A*" meaning that the value of A is to be placed at the top of a pushdown stack. The symbol "+" is the binary machine operator "*ADD*," which causes the contents of the top two cells in the pushdown stack to be added and their sum placed in a single cell that replaces the original pair. In the latter case the stack has been "popped up." In order to allow the translator as much freedom as possible in determining how to group expressions, expressions containing subtraction operations will be compiled with the unary operator *MINUS*. Thus, the expression $A + B - C + D$, which cannot correctly be parenthesized as $(A + B) - (C + D)$, will be translated as if it were the expression $(A + B) + ((-C) + D)$. The suffix Polish form of the latter is AB + $C - D$ ++. Similarly, the divide operator "/" will be treated as a unary *RECIPROCATE* operator to permit the expression $A \times B/C \times D = (A \times B \times D)/C$ to be translated to the equivalent of $(A \times B \times ((1/C) \times D))$, i.e., $AB \times C/D \times\times$ in suffix Polish.

As given here, the translation algorithm will not recognize expressions containing a true unary operator such as the negation in "$-A + B$," even though unary *MINUS* operations are produced as output by the algorithm. No attempt has been made to include this capability in the algorithm in order to leave it as uncluttered as possible.

3. Compiler Description

Since the translator processes are intimately connected with the syntax underlying the language of the arithmetic expressions, we summarize the language accepted by the translator here. The language description is in modified Backus Normal Form [5] which uses the metasymbols "::=" to mean "is defined to be," "⟨" and "⟩" to bracket syntactic units, "|" to mean "or," and the metabraces and star such that "{X}*" means "empty | X | XX | XXX | ⋯ ." The language of the expressions can be described by the five syntactic units—variable, primary, factor, term, and expression—as follows:

```
⟨expression⟩ ::= ⟨term⟩ {⟨adding operator⟩ ⟨term⟩}*
⟨term⟩       ::= ⟨factor⟩ {⟨multiplying operator⟩ ⟨factor⟩}*
⟨factor⟩     ::= ⟨primary⟩ {↑⟨primary⟩}*
⟨primary⟩    ::= ⟨variable⟩ | (⟨expression⟩)
⟨variable⟩   ::= A | B | C | ⋯ | Z
```

where the adding operators are the operators "+" and "−", the multiplying operators are "×" and "/", and the "↑" denotes exponentiation. The syntactic unit ⟨variable⟩ denotes a simple variable.

From this point we shall describe the compilation technique in general and leave the details to a set of Algol procedures that are found in the Appendix. The reader is urged to use the syntactic description of the expression language as a guide to understanding the Algol text.

Since the compiler must scan an input string, it is assumed that there is a procedure called *SCAN* that keeps track of the position of the scanner in the input string and produces a new elementary syntactic unit while stepping the input scanner each time it is called. The new syntactic unit is assumed to be stored in the integer variable called *ITEM*. Here we make the assumption that the character is scanned and converted using the *A* format as described in [8] so that we can make use of the built-in procedure *EQUIV* for converting string characters to their internal integer representation. The procedure *OUTPUT* is assumed to convert its argument from integer representation to a string character and place it in the output string.

In order to generate a tree-like hierarchy of operations, the scanning technique keeps track of the tree level and places the operators in the output string accordingly. To see how the procedures automatically take care of the necessary bookkeeping, consider how the expression $A + B + C + D + E + F + G + H$ is compiled. For the moment, only the procedures *EXPRESSION* and *TERM* are pertinent, and we shall assume that *TERM* has been modified to read:

```
if level = 0 then
  begin output (item);
    scan;
  end
```

etc., in place of the calls on *FACTOR*, which are listed in the Algol text.

We use an inductive argument to show that a binary tree is produced for the example expression. Induction is on the value of *LEVEL* that is set by *EXPRESSION* for a call on *TERM*. The first call on *TERM* with *LEVEL* = 0, produces the output symbol "A" and leaves the next operator "+" in *ITEM*. The output is a binary tree with one node, height 0. *EXPRESSION* calls *TERM* at level 1 because *ITEM* contains a "+". At this level, *TERM* moves the scanner forward to place "B" in *ITEM*, calls itself at the 0th level, and places a "+" in the output string before exiting back to *EXPRESSION*. Since the call on *TERM* at level 0 will compile a binary tree with one node, the output string has become "$AB+$" and the scanner has moved forward to place the second "+" operator in *ITEM*. Note that the iterative call on *TERM* by *EXPRESSION* has resulted in the compilation of a binary tree with two nodes, and of height 1.

Proceeding inductively, assume that a binary tree of height $n-1$ has been generated by n-iterated calls on *TERM*, each at a successively higher level. At the nth exit of *TERM* back to *EXPRESSION*, *ITEM* will contain the operator "+" if there is one left uncompiled in the input string. The call on *TERM* at level n will result in a scan to place a new operand in *ITEM*, an iterated sequence of n calls on *TERM* at successively higher levels beginning at level 0, and the insertion of a "+" in the output string. Since the internally generated sequence of calls on *TERM* are identical to the calls generated by *EXPRESSION*, they will result in the compilation of another binary tree of height $n-1$. Since the operator "+" is suffixed to two binary trees of height $n-1$, the result is a full binary tree of height n. Furthermore, *ITEM* will contain the next uncompiled operator in the input string.

A few words of explanation are necessary to fill in the program details. The action of *TERM* and *FACTOR* with respect to the compilation of terms is analogous to the action of *EXPRESSION* and *TERM*. Hence, the procedure *FACTOR* is almost identical to *TERM*. The operator " ↑ " cannot be treated as the multiplying and adding operators have been treated because it is not associative. Hence, *FACTOR* produces the string $AB\uparrow C\uparrow D\uparrow$ from the expression $A\uparrow B\uparrow C\uparrow D$ without reordering the operators in a treelike structure.

Expressions within parentheses are compiled as entities within an output string by the recursive call on *EXPRESSION* contained within procedure *PRIMARY*. Notice in particular how the action of the variables *MINUS* and *DIVIDE* control the code emitted for the unary operators while a recursive call on *EXPRESSION* is in effect. Normally when a unary operator is scanned, the corresponding Boolean variable *DIVIDE* or *MINUS* is set **true**. At Level 0, *TERM* and *FACTOR* check these variables to determine if a unary operator ought to be emitted. In case *PRIMARY* scans an expression between the setting of a Boolean variable and the testing of the variable, the values of the variables are "pushed-down" while the recursive call of *EXPRESSION* is in effect. The recursive call of *EXPRESSION* is begun with **false** values of *DIVIDE* and *MINUS*, and at the termination of the call when the closing parenthesis is scanned, the old values of *MINUS* and *DIVIDE* are restored.

Inspection of the following examples should clarify the description of the compiler.

Expression	*Compiled Expression*
$A+B+C+D+E+F+G+H$	$AB+CD++EF+GH+++$
$A+B+C+D\times E\times F+G+H$	$AB+CDE\times F\times ++GH++$
$A+B-C-D\times E\times F+G+H$	$AB+C-DE\times F\times -++$ $GH++$
$A+B-C-D\times (E\times F+G+H)$	$AB+C-DEF\times G+$ $H+\times -++$
$A+B-C-D/(E\times F+G\uparrow H)$	$AB+C-DEF\times GH\uparrow +/$ $\times -++$
$A+B-C-D+(E+F\uparrow G\times H/$ $I\times J\times K)/L+M+N$	$AB+C-D-++EFG\uparrow H\times I/$ $J\times\times K\times +L/\times M+N++$

Acknowledgment. The author is indebted to M. W. Green and C. L. Jackson of Stanford Research Institute and to the referees for several comments and suggestions that have been incorporated in this paper.

RECEIVED SEPTEMBER, 1966; REVISED DECEMBER, 1966

APPENDIX. Compiler Procedures (In ALGOL)

```
procedure expressions;
begin
integer item;
Boolean minus, divide;
integer level;
procedure term(level);  value level;  integer level;
begin integer treelevel;
  if level = 0 then
  begin factor (0);
    treelevel := 1;
    for treelevel + 1 while item = equiv('×') ∨ item = equiv('/')
      do factor(treelevel);
    if minus then outstring(1, '−');
    minus := false:
  end
  else
  begin
    minus := item = equiv('−');
    scan;
    term(0);
    treelevel := 0;
    for treelevel := treelevel + 1 while (item = equiv('+') ∨ item
      = equiv('−')) ∧ treelevel < level do term(treelevel);
    output('+')
  end
end term;
procedure factor(level);  value level;  integer level;
begin integer treelevel;
  if level = 0 then
  begin primary;
    treelevel := 0;
    for treelevel := treelevel + 1 while item = equiv('↑') do
      begin scan;
        primary;
        outstring(1, '↑')
      end
    if divide then outstring(1, '/')
    divide := false;
  end
  else
  begin divide := item = equiv('/');
    scan;
    factor (0);
    treelevel := 0;
    for treelevel := treelevel + 1 while (item = equiv('×') ∨ item
      = equiv('/')) ∧ treelevel < level do factor(treelevel);
    outstring(1, '×')
  end
end factor;
procedure primary;
begin
  if item = equiv('(') then
  begin expression;
    if item ≠ equiv(')') then error else scan
  end
  else
  begin output(item);
    scan
  end
  end primary;
comment  the body of expression begins here;
  divide := minus := false;
  term(0);
  level := 0;
  for level := level + 1 while item = equiv('+') ∨ item =
    equiv('−') do
    term(level)
end expression
```

REFERENCES

1. Hamblin, C. L. Translation to and from Polish notation. *Comput. J. 5* (Oct. 1962), 210–213.
2. Allard, R. W., Wolf, K. A., and Zemlin, R. A. Some effects of the 6600 computer on language structures. *Comm. ACM 7*, 2 (Feb. 1964), 112–119.
3. Hellerman, H. Parallel processing of algebraic expressions. *IEEE Trans. EC-15*,1 (Feb. 1966), 82–90.
4. Squire, J. S. A translation algorithm fos a multiple processor computer. Proc. 18th ACM Nat. Conf., Denver, Colorado, 1963.
5. Carr, III, J. W., and Weiland, J. A nonrecursive method of syntax specification. *Comm. ACM 9*, 4 (April 1966), 267–269.
6. Revised report on the algorithmic language Algol 60. *Comm. ACM 6*,1 (Jan. 1963), 1–17; see Sec. 3.3.5, pp. 7–8.
7. FORTRAN vs. BASIC FORTRAN. *Comm. ACM 7*, 10 (Oct. 1964), 590–625; see Sec. 6.4, pp. 598–599.
8. A proposal for input-output conventions in Algol 60. *Comm. ACM 7*, 5 (May 1965), 273–283; see Sec. A.2.3.2, p. 275.

A NOTE

Top-to-Bottom Parsing Rehabilitated?

R. A. Brooker
Manchester University, Manchester, England*

This note is concerned with the efficiency of the Top-to-Bottom parsing algorithm as used in connection with programming language grammars. It is shown, for instance, that retracing of unprofitable paths can often be eliminated by a suitable rearrangement of the productions defining the grammar. The essential weakness of the method is in dealing with complicated syntactic structures which are in practice only sparsely occupied, e.g., arithmetic expressions.

The question is sometimes raised as to the relative merits of syntax analysis "top down" and "bottom up" (see, e.g., the Discussion following Leavenworth [1]). There seems to be little published evidence.

Griffiths and Petrick [2] remark (in a paper on the relative efficiencies of context-free grammar recognizers), "In this comparison we found our SBT procedure to be enormously more efficient than our STB procedure for the Lisp and Algol programming language grammars considered, and generally superior for all other grammars considered except those for which the recognizers were deterministic."

While not doubting their conclusions *for the particular grammars they considered* (although even the authors themselves admit to some discrepancy between some of their conclusions and experience obtained in the field), it is the purpose of this note to draw attention to the remarks found in Cheatham [3]: "For programming languages of the current sort, there is no clear advantage in favor of either the top-down or bottom-up analysis techniques, insofar as efficiency of the analyzer is concerned. For either technique, it is possible to design a language and syntax specification on which the technique will perform very poorly, while the other one will not be nearly so bad. The choice between the techniques is generally made on the basis of considerations other than raw speed of the analysis, ...".

Now in [2] only one grammar is presented in detail which supports the authors' conclusions. It is

$$\begin{array}{ll} F \to C & L \to L' \\ F \to S & L \to p \\ F \to P & L \to q \\ F \to U & L \to r \\ C \to U \supset U & S \to U \vee S \\ U \to (F) & S \to U \vee U \\ U \to \neg U & P \to U \wedge P \\ U \to L & P \to U \wedge U \end{array}$$

The following sentence is one which they parse w.r.t. this grammar $\neg(\neg(p' \wedge (q \vee r) \wedge p'))$. If we write the grammar in the more concise form

$$\begin{array}{l} F \to C \mid S \mid P \mid U \\ C \to U \supset U \\ U \to (F) \mid \neg U \mid L \\ L \to L' \\ L \to p \mid q \mid r \\ S \to U \vee S \mid U \vee U \\ P \to U \wedge P \mid U \wedge U \end{array}$$

it will be clear why recognizing the above sentence could involve a vast amount of retracing. Thus top-to-bottom starts by looking for a "C", which means looking for a "U", then a "(", which it does not find, then "$\neg U$", which after a fantastic search it eventually finds; *then* it returns to the C-production and looks for a "$\supset$", which *it does not find*, and so returns to the F-production and starts looking for an "S" instead, and so on.

The authors of [2] remark, "In order to determine the extent to which the disparity in efficiency between the

* Department of Computer Science