WIS-CS-75-243

COMPUTER SCIENCES DEPARTMENT
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

March 13, 1975

# ON PARSING AND COMPILING ARITHMETIC EXPRESSIONS IN PARALLEL COMPUTATIONAL ENVIRONMENTS

by

Charles N. Fischer

Technical Report #243

MARCH 1975

## ABSTRACT

The problem of parsing and compiling arithmetic expressions in parallel computational environments is considered. It is seen that the concept of Operator Precedence can be generalized to allow encodings of one or more arithmetic expressions to be transformed directly into encodings of their corresponding derivation trees. The algorithm which performs this transformation is compact, efficient (linear in both time and space), and highly concurrent. Further, it can be extended to compile arithmetic expressions directly into object code (in the form of quadruples). The extention preserves the compactness, efficiency (linearity) and highly concurrent nature of the original algorithm.

## 1. Introduction

With the advent of parallel computers such as the Illiac IV and CDC Star-100, the design and analysis of algorithms which utilize parallelism has become of great interest. Rather surprisingly, however, the problem of designing compilers to run on such parallel computers has received relatively little attention ([1],[3],[4]). In this paper we consider the problem of parsing and compiling arithmetic infix expressions in parallel environments. Such expressions are of interest in that they are an integral part of virtually all programming languages; in fact in some languages (such as APL) they represent the only non-trivial structure present. We suggest it is entirely plausible that on suitable parallel computers all the arithmetic expressions occurring in a program segment, or perhaps an entire program, might be parsed (or compiled) concurrently in an efficient manner.

First we introduce Arithmetic Infix Grammars (AIG's) as a means of describing the class of arithmetic infix expressions. Next it is seen that a number of arithmetic expressions can be parsed efficiently and with a high degree of concurrency. In particular, encodings of such expressions can be transformed directly into encodings of their corresponding derivation trees. This enables us to avoid the overhead, inherent in serial parsing techniques, of repeatedly finding and replacing single occurrences of various productions. Finally, this parsing technique is extended to allow arithmetic expressions to be compiled directly into object code (in the form of quadruples). This extension allows us to by-pass conventional parsing entirely and yet is still both efficient and highly concurrent.

## 2. Arithmetic Infix Grammars

The structure of an arithmetic infix expression is determined by the properties of its operators. Usually the set of operators used is partitioned into a number of classes. Each operator class is distinguished by its priority (often called precedence) and its associativity (either right or left).

Thus a given Arithmetic Infix Language will be characterized by $n$ pairwise disjoint operator classes $OP_1,\ldots,OP_n$. All unary operators will be placed in $OP_n$. Operators in $OP_{i+1}$ will have a higher priority than those in $OP_i, OP_{i-.}$, etc. The highest priority operators (the unary ones) are applied first, then the next highest, etc. Further, each operator class, $OP_i$, has an associativity, $ASSOC[i]$. $-1$ denotes left associativity; $1$ denotes right associativity. By definition, $ASSOC[n] = 1$ (right). Finally, we shall assume a class $OP_0 \equiv \{\#\}$ is added. # is a null operator used as an endmarker and to separate adjacent arithmetic expressions. We assume $ASSOC[0] = -1$.

For example, an Arithmetic Infix Language having the operators +,-,*,/,**,SQRT and the usual semantics would be characterized as follows:

$$OP_0 = \{\#\},\ OP_1 = \{+,-\},\ OP_2 = \{*,/\},\ OP_3 = \{**\},\ OP_4 = \{SQRT\}$$

$$ASSOC[0] = -1,\ ASSOC[1] = -1,\ ASSOC[2] = -1,\ ASSOC[3] = 1,\ ASSOC[4] = 1$$

We may now define an <u>Arithmetic Infix Grammar</u> which generates the Arithmetic Infix Language characterized by $OP_0,\ldots,OP_n$ and $ASSOC[0],\ldots,ASSOC[n]$ as $G = (V, V_T P, S)$.

$$V_T = \{ID,(,)\} \cup OP_0 \cup \ldots \cup OP_n$$

$$V = V_T \cup \{S, N_0, \ldots, N_{n+1}\}$$

$P$ is the union of the following sets of productions:

(1) $\{S \rightarrow \#N_i\# \mid 0 \leq i \leq n+2\}$

(2) $\{N_n \rightarrow \theta N_i \mid \theta \varepsilon OP_n$ and $n \leq i \leq n+2\}$

(3) $\{N_{n+1} \rightarrow (N_i) \mid 1 \leq i \leq n+2\}$

(4) For $0 \leq j \leq n-1$ :

If $ASSOC[j] = -1$

then $\{N_j \to N_i \theta N_k | \theta \varepsilon OP_j$ and $j \leq i \leq n+2$ and $j < k \leq n+2\}$

else $\{N_j \to N_i \theta N_k | \theta \varepsilon OP_j$ and $j < i \leq n+2$ and $j \leq k \leq n+2\}$

Note that, by definition, $N_{n+2} \equiv ID$ .

Theorem 1. If G is an AIG then G is unambiguous.

Proof: It is easy to establish that G is LR(1) . ∎

## 3. Parsing Arithmetic Infix Expressions

In general a parser serves two functions. It verifies that an input string is, in fact, in L(G) (that is, derivable from S) and, given that it is, it produces a parse of it.

If G is an AIG , then, as we shall see, it is especially simple to test whether an input string is in L(G) . First let $U \subseteq V_T$ for some AIG . Then we can define $Follow(U) = \{b \varepsilon V_T | S \to^+ ...ab... \text{ and } a \varepsilon U\}$ . Follow(U) is the set of terminal symbols that may legitimately follow some symbol in U .

Let UN (the set of unary operators) = $OP_n$ and BIN (the set of binary operators) = $OP_0 \cup ... \cup OP_{n-1}$ . It can easily be verified that the Follow sets listed below are correct.

| Set: | UN | BIN | {(} | {)} | {ID} |
|---|---|---|---|---|---|
| Follow: | W | W | W | W' | W' |

where $W \equiv UN \cup \{(,ID\}$ and $W' \equiv BIN \cup \{)\}$ .

Next, let $a \varepsilon V_T$ and $x \varepsilon V_T^*$ . Then COUNT(x,a) is a count of the number of occurrences of a in x . For example, COUNT(aabaa,a) = 4 and COUNT(aabaa,b) = 1 .

Finally, let $x \varepsilon V_T^+$ for G some AIG . Then x is Well Formed (with respect to G) iff:



(1) x is of the form #x'#

(2) $x = x_1x_2 \Rightarrow COUNT(x_1,() - COUNT(x_1,)) \geq 0$ and $x = x_3\#x_4 \Rightarrow COUNT(x_3\#,() - COUNT(x_3\#,)) = 0$ for $x_1,x_2,x_3,x_4 \varepsilon V_T^*$ .

(3) $x = ...ab... \Rightarrow b \varepsilon Follow(\{a\})$ .

Note that condition (1) requires correct endmarkers and that condition (2) requires proper parenthesis structure. The following establishes the importance of Well Formedness:

Theorem 2. Let G be an AIG . Then $x \varepsilon L(G)$ iff x is Well Formed.

Proof: Clearly $x \varepsilon L(G) \Rightarrow x$ is Well Formed. Assume x is Well Formed and write x as $\#x_1\#x_2\#...\#x_m\#$ where # does not occur in $x_1,...,x_m$ . Write $x_1$ as ...(x')... where x' contains no "(" or ")"; otherwise, let $x' = x_1$ . Using Follow sets, x' must be of the form $(UN^* Z)(BIN\ UN^* Z)^*$ where $Z = \{ID,N_{n+1}\}$ . Unary operators in x' can be reduced to obtain x" which is of the form $Y(BIN\ Y)^*$ where $Y = \{ID,N_{n+1},N_n\}$ . A simple induction on the number of operators in x" will establish that $N_i \to^* x"$ for $1 \leq i \leq n+2$ (recall $ID \equiv N_{n+2}$) . Now if we had $x' = x_1$ then we have established that $N_i \to^* x_1$ ; otherwise we have reduced $x_1 = ...(x')...$ to $...(N_i)...$ But $...(N_i)...$ can be reduced to $...N_{n+1}...$ and the above process is then repeated. Thus finally $x_1$ is reduced to $N_i$ for $1 \leq i \leq n+2$ . $x_2,...x_m$ can be reduced in like manner. Thus x can be reduced to $\#N_i\#N_j\#...\#N_q\#$ for $1 \leq i,j,...,q \leq n+2$ . It is then easy to verify that $S \to^+ \#N_i\#N_j\#...\#N_q\#$ . ∎

Note that Well Formedness can easily be tested in linear time. In fact conditions (1) and (2) require but a single scan of the input while condition (3) can be tested in a highly concurrent manner. We may now attack our main objective - that of parsing arithmetic infix expressions.

The order of application of operators in an expression depends on a number of factors:

(1) The depth of the parenthesis nesting

(2) The priority of the operators

(3) The associativity of the operators

(4) The position of the operators in the expression

We shall encode these factors, uniquely, for each operator, in an integer. This integer, the <u>precedence</u> of the operator, is an extention of the concept of Operator Precedence ([2]).

Let $INPUT = a_1 \ldots a_p$ be a Well Formed input string. Further assume $OPS = \theta_1 \ldots \theta_q$ are the operators occurring in INPUT and that $INDEX = i_1, \ldots, i_q$ contains the positions of the operators in INPUT (that is, OPS = INPUT [INDEX]). Finally, if $\theta$ is an operator then let $CLASS(\theta) = j$ iff $\theta \varepsilon OP_j$ .

We can now encode the four factors noted above:

(1) $NEST(\theta_j) = COUNT(a_1 \ldots a_{i_j}, () - COUNT(a_1 \ldots a_{i_j}, ))$

$NEST(OPS) = NEST(\theta_1), \ldots, NEST(\theta_q)$

(2) $PRIO(OPS) = CLASS(\theta_1), \ldots, CLASS(\theta_q)$

(3) $ASSOCIATIVITY(OPS) = ASSOC[CLASS(\theta_1)], \ldots, ASSOC[CLASS(\theta_q)]$

(4) $POS(OPS) = 1, \ldots, q$

Finally, we obtain PREC(OPS) = (POS(OPS)*ASSOCIATIVITY(OPS)) + 2*(q+1)*PRIO(OPS) + 2*(q+1)*(n+1)*NEST(OPS) where q = |OPS| and n = (as usual) the number of operator classes.

PREC(OPS) encodes the type, position and parenthesis level of each operator in OPS. It should be clear that (given q and n) the encoding is unique for it can easily be uniquely inverted.

NEST can be calculated by a single scan over INPUT while PRIO, ASSOCIATIVITY, POS and PREC can be calculated in a highly concurrent manner. Clearly the entire calculation is linear.

As an example, using the same operator classes as before, let

INPUT = #ID*(ID + ID**ID) - SQRT ID#SQRT((ID + ID)/ID)#

Then



OPS = #,*,+,**,-,SQRT,#,SQRT,+,/,#

INDEX = 1,3,6,8,11,12,14,15,19,22,25

NEST(OPS) = 0,0,1,1,0,0,0,0,2,1,0

PRIO(OPS) = 0,2,1,3,1,4,0,4,1,2,0    POS(OPS) = 1,...,11

ASSOCIATIVITY(OPS) = -1,-1,-1,1,-1,1,-1,1,-1,-1,-1    q = 11, n = 4

PREC(OPS) = -1,46,141,196,19,102,-7,104,255,158,-11

Given the derivation tree of an arithmetic infix expression we may define a derivation tree (called a <u>reduced derivation tree</u>) which involves only ID's and operators as follows:

(1) Replace each $N_i$ $(0 \le i \le n)$ by the operator it derives.

(2) Replace subtrees of the form

```
   N
    n+1
  / | \          by   T1    and     / | \       by   T2
 (  T1  )                          #  T2  #
```

Thus the reduced derivation tree corresponding to the above example is:

```
                     # (-7)
                   /      \
              - (19)        SQRT (104)
             /     \              \
      * (46)    SQRT (102)        / (158)
     /   \            \          /      \
   ID   + (141)        ID     + (255)    ID
        /   \                  /  \
      ID    ** (196)          ID  ID
            /  \
          ID    ID
```

Observe that the PREC values have the property that <u>all</u> operators occurring in a subtree rooted by $\theta$ have a <u>larger</u> PREC value than that of $\theta$ . This, of course, is by no means accidental.

<u>Theorem 3</u>. Let some operator $\theta_i$ occur in a subtree rooted by the operator $\theta_j$ in the reduced derivation tree of some arithmetic infix expression. Then PREC[i] > PREC[j].

<u>Proof</u>: Consider the corresponding (non-reduced) derivation tree. Since this tree is generated by some AIG, it must be the case that:
(1) The parenthesis level of $\theta_i$ is $\geq$ than that of $\theta_j$ .

(2) If the parenthesis levels of $\theta_i$ and $\theta_j$ are equal then $\mathrm{CLASS}(\theta_i) \geq \mathrm{CLASS}(\theta_j)$

(3) If $\mathrm{CLASS}(\theta_i) = \mathrm{CLASS}(\theta_j)$ then

(a) If $\mathrm{ASSOC}[\mathrm{CLASS}(\theta_i)] = -1$ then $\theta_i$ is to the left of $\theta_j$ , that is, $i < j$

(b) If $\mathrm{ASSOC}[\mathrm{CLASS}(\theta_i)] = 1$ then $\theta_i$ is to the right of $\theta_j$ , that is, $i > j$ .

But these conditions ensure that (by the definition of PREC) that PREC[i] > PREC[j] . ∎

The reduced derivation tree of an expression contains almost exactly the same structure as the corresponding non-reduced derivation tree (only non-terminals, endmarkers and parentheses are lost). Thus we will consider the problem of parsing an expression to be that of determining its reduced derivation tree. The following functions are key to such a determination.

Let Q be a vector of unique integers. Then define:

$$\mathrm{SR}(i,Q) = \{j \mid i < j < |Q| \text{ and } \neg\exists k(i < k \leq j \text{ and } Q[k] < Q[i])\}$$

$$\mathrm{SL}(i,Q) = \{j \mid 1 < j < i \text{ and } \neg\exists k(j \leq k < i \text{ and } Q[k] < Q[i])\}$$

SR <u>s</u>cans <u>r</u>ight from i , getting the largest set of consecutive integers having Q values > Q[i] . SL does the same, <u>s</u>canning <u>l</u>eft. We then define



$$\mathrm{L}(i,Q) = \text{If } \mathrm{SL}(i,Q) = \emptyset \text{ then } 0 \text{ else } j\in \mathrm{SL}(i,Q) \text{ st } \forall k\in \mathrm{SL}(i,Q)\ Q[j] \leq Q[k]$$

$$\mathrm{R}(i,Q) = \text{If } \mathrm{SR}(i,Q) = \emptyset \text{ then } 0 \text{ else } j\in \mathrm{SR}(i,Q) \text{ st } \forall k\in \mathrm{SR}(i,Q)\ Q[j] \leq Q[k]$$

L (R) simply finds that index in SL (SR) that has the smallest Q value, or returns 0 . The import of these functions is established in the following.

<u>Theorem 4</u>. Let INPUT $\in$ L(G) for G some AIG and let PREC(OPS) be as above.

Then (1) SR(i,PREC(OPS)) (SL(i,PREC(OPS))) = the indices of all the operators in the right (left) subtree of the i-th operator.
(2) R(i,PREC(OPS)) (L(i,PREC(OPS))) = the index of the operator that roots the right (left) subtree of the i-th operator if it exists, otherwise 0 .

<u>Proof</u>: (1) Consider SR (SL is analogous). All operators in $\theta_i$'s right subtree are to the immediate right of $\theta_i$ and by Thm. 3 their indices will be included in SR(i,PREC(OPS)) . Let $\theta_j$ $(j > i)$ be the first operator in the reduced derivation tree that is to the right of $\theta_i$ but <u>not</u> in $\theta_i$'s right subtree. It must be that PREC[i] > PREC[j] (if $\theta_i$ were not a descendent of $\theta_j$ then $\theta_i$ and $\theta_j$ would have a common ancestor lying between them). Thus SR(i,PREC(OPS)) is exactly the set of indices of the operators in $\theta_i$'s right subtree.

(2) If SR(i,PREC(OPS)) = $\emptyset$ then no operators are in $\theta_i$'s right subtree and R(i,PREC(OPS)) = 0 as required. Otherwise, the root of $\theta_i$'s right subtree is (by Thm. 3) that operator in the subtree with the minimum PREC value. But this is just R(i,PREC(OPS)) . A similar argument holds for L(i,PREC(OPS)) . ∎

Observe that the L and R functions are just what we need to determine the reduced derivation tree. R(i,PREC(OPS)) , for example, either points to the root of $\theta_i$'s right subtree or tells us (by returning 0) that the first ID to $\theta_i$'s right is the correct right subtree.

We are now ready to present an AIG parser.

Algorithm 5. (An AIG parser)

Input: One or more arithmetic infix expressions, separated by #'s, stored in INPUT.

Output: Encodings of the reduced derivation trees of the arithmetic infix expressions stored in EXPR_ROOTS, LEFT_SUBTREE, RIGHT_SUBTREE.

[1] If INPUT does not satisfy conditions (1), (2) and (3) of the Well Formedness requirement then stop and signal an error in INPUT.

[2] Calculate PREC(OPS).

[3] Let #_INDEX = the indices of all but the rightmost # in OPS
(That is OPS[#_INDEX] = #,#,...,#)
Let UNARY_INDEX = the indices of all unary operators in OPS
LEFT_SUBTREE = L(1,PREC(OPS)),...,L(|OPS|-1,PREC(OPS))
LEFT_SUBTREE[UNARY_INDEX] = -1
RIGHT_SUBTREE= R(1,PREC(OPS)),...,R(|OPS|-1,PREC(OPS))
EXPR_ROOTS = RIGHT_SUBTREE[#_INDEX] ∎

The encodings of the reduced derivation trees are quite straightforward. Recall that INPUT is of the form $\#E_1\#E_2\#\ldots\#E_r\#$ where $E_1,\ldots,E_r$ are individual arithmetic expressions. In particular, $E_i$ is delimited by the i-th and i+1 st #'s. By construction the PREC value of the i-th # is < the PREC values of all the operators in $E_i$ and is > the PREC value of the i+1st #. This means that if the R value of the i-th # is j then if j > 0 then $\theta_j$ is the root of $E_i$'s reduced derivation tree and if j = 0 then $E_i$ is a single ID to the immediate right of the i-th # .

To find the roots of the reduced derivation trees of $E_1,\ldots,E_r$



we then simply look up EXPR_ROOTS[1],...,EXPR_ROOTS[r]. Further, if $\theta_j$ is an operator $\neq$ # then if RIGHT_SUBTREE[j] $\neq$ 0 then it points to the operator that is the root of $\theta_j$'s right subtree. If RIGHT_SUBTREE[j] = 0 then $\theta_j$'s right subtree is the first ID to the right of $\theta_j$ . So too, if $\theta_j$ is binary then LEFT_SUBTREE[j] either points to the operator that is the root of $\theta_j$'s left subtree or indicates that $\theta_j$'s left subtree is the first ID to $\theta_j$'s left. LEFT_SUBTREE[j] is = -1 if $\theta_j$ is unary.

Reconsidering the example used earlier, we had

INPUT = #ID*(ID+ID**ID) - SQRT ID#SQRT((ID+ID)/ID)#
PREC(OPS) = -1,46,141,196,19,102,-7,104,255,158,-11.

If we apply Algorithm 5 we get:

LEFT_SUBTREE = 0,0,0,0,2,-1,5,-1,0,9
RIGHT_SUBTREE = 5,3,4,0,6,0,8,10,0,0
EXPR_ROOTS = 5,8

The reader may easily verify that the reduced derivation trees of the two expressions are correctly encoded.

Theorem 6. Let G be some AIG. Then

(1) If INPUT $\varepsilon$ L(G) then Alg 5 produces a correct encoding of the reduced derivation tree(s) of the arithmetic expression(s) in INPUT. Further, if the L and R functions are evaluated in O(|INPUT|) time and space then Alg 5 will execute in O(|INPUT|) time and space.

(2) If INPUT $\notin$ L(G) then Alg 5 will signal on error and stop.

Proof: (1) Correctness: Follows from Thm 2 and Thm 4.
Linearity: As noted earlier steps [1] and [2] can be done in linear time and space and by assumption so can step [3].

(2) Follows from Thm 2. ∎

At this point we reemphasize the fact that Alg 5 transforms an encoding of the input expressions (the PREC vector) <u>directly</u> into an encoding of the corresponding reduced derivation trees (the EXPR_ROOTS, LEFT_SUBTREE, and RIGHT_SUBTREE vectors). The role of the AIG is purely descriptive. Thus the usual overhead of finding and replacing occurrences of productions is avoided.

## 4. Calculating the L and R Functions

We now consider the problem of calculating L and R. For convenience let $\bar{L}(Q)$ denote $L(1,Q),\ldots,L(|Q|-1,Q)$ and similarly for $\bar{R}(Q)$. Clearly $\bar{L}(Q)$ and $\bar{R}(Q)$ are closely related, in fact

$$\bar{L}(Q) = 0,(|Q|-\bar{R}^{REV}(Q[|Q|-1,\ldots,1]))\ \mathrm{Mod}\,|Q|$$

where $V^{REV}$ denotes $V$ <u>reversed</u>.

Note that the Mod function is used solely to insure that the integers in $\bar{L}(Q)$ are in the range $|Q|-1$ to $0$.

Thus we need consider only methods of calculating $\bar{R}(Q)$. One method is the following*.

### Algorithm 7

Input: A vector $Q$ of unique integers
Output: $\bar{R}(Q)$ stored in RESULT

```
[1]  Stack (-∞,-∞) onto an empty push down stack where doublets
     are of the form
     (STK_POINTER,STK_Q)
     POINTER ← 1; Q[|Q|] ← -∞
[2]  Do while POINTER ≤ |Q| :
     If  STK_Q(top stack element) ≤ Q[POINTER]
     then   stack (POINTER,Q[POINTER])
            POINTER ← POINTER + 1
            PREV ← 0
     else RESULT[STK_POINTER(top stack element)] ← PREV
            PREV ← STK_POINTER(top stack element)
            pop off the top doublet on the stack           ∎
```

*this algorithm was originally suggested by John Hopcroft



<u>Theorem 8</u>. Given input $Q$ Alg 7
(a) Computes $\bar{R}(Q)$ correctly,
(b) In time and space bounded by $O(|Q|)$.

<u>Proof</u>: (a) When POINTER = $r$ for $1 \leq r < |Q|$, $(r,Q[r])$ is stacked (after perhaps popping a number of stack elements). $(r,Q[r])$ remains on the stack until POINTER = $j$ where $Q[j] < Q[r]$. That is, $j$ is the first index <u>not in</u> $SR(r,Q)$. Thus $r+1,\ldots,j-1$ are the elements of $SR(r,Q)$. Each has already been considered, including that value $k$ $(r+1 \leq k \leq j-1)$ having the minimum $Q$ value (if $SR(r,Q) \neq \emptyset$). By definition $k = R(r,Q)$. Further, it will be stacked immediately above $(r,Q[r])$ since $r+1,\ldots,k-1$ have $Q$ values larger than $Q[k]$. Thus when $(r,Q[r])$ is popped, PREV $= k = R(r,Q)$. Also, if $j = r+1$, then $SR(r,Q) = \emptyset$ and PREV $= 0 = R(r,Q)$.

(b) During each iteration of step [2] an element of vector $Q$ is pushed or popped.

Further, each element of $Q$ is pushed and popped at most once. ∎

It may appear that the above algorithm is too "serial" to utilize parallel environments efficiently. However this need not be the case. Recall that in general the input to our parser will be of the form $\#E_1\#E_2\#\ldots\#E_m\#$. For a given $E_r$ $(1 \leq r \leq m)$ <u>all</u> its L and R values will of necessity lie within $E_r$. Further, the R value of all but the rightmost # points to the root of the expression to its immediate right. The L values of #'s are not used and need not be computed. Thus the input may be divided into a number of independent segments $\#E_1\#,\#E_2\#,\#E_m\#$. Alg 7 may then be run concurrently on each of these segments. In fact this approach is similar to that taken by Zozel, et al [4] in performing a conventional Operator Precedence parse in parallel.

We now consider an algorithm which is "more parallel" in structure. Let a REPEAT b = the vector b,b,...,b of length a . Further, let B be a boolean vector and V,W,V' be arbitrary vectors all of the same length. Then V,W MASK B = V' where for $1 \le r \le |V|$ V'[r] = if B[r] = 1 then W[r] else V[r] . Thus (1,2,3,4),(5,6,7,8) MASK 0,1,1,0 = 1,6,7,4 . (Note that REPEAT and MASK are actual STAR-100 instructions.)

Algorithm 9.

Input: Q a vector of unique integers and LIMIT < |Q|

Output: A vector RESULT and a bit vector HAVERESULT.

```
[1]  BIG ← any integer > Max(Abs(Q))
     R_RANGE ← 1,...,|Q|-1
     QI ← (R_RANGE + |Q|*Q[R_RANGE]) CONCAT
          (LIMIT REPEAT (|Q|*-BIG))
     BIGVECTOR ← MINVAL ← (|Q|-1) REPEAT (|Q|*BIG)

     HAVERESULT ← (|Q|-1) REPEAT 0
[2]  For POS from 1 to LIMIT do:
     HAVERESULT ← HAVERESULT v(QI[POS+R_RANGE]
                  < QI[R_RANGE])
     COMPARE_VAL ← QI[POS+R_RANGE],BIGVECTOR
                     MASK HAVERESULT
     MINVAL ← Min(MINVAL,COMPARE_VAL)
[3]  RESULT ← MINVAL mod |Q|                       ∎
```

Theorem 10. Given Q and LIMIT as inputs to Alg 9,

(a) If HAVERESULT[r] = 1 then RESULT[r] = R(r,Q) .

(b) The algorithm is bounded in space by O(|Q|) and in time by O(LIMIT*|Q|) .

Proof: (a) For each index r ($1 \le r < |Q|$) we encode r and Q[r] into QI[r] in such a way that QI[r] < QI[j] iff Q[r] < Q[j] . MINVAL and BIGVECTOR are initialized to an encoding of a very large Q value and a 0 index. In step [2], HAVERESULT[r] is set to 1 the first time that Q[r+POS] < Q[r] . Thus SR(r,Q) = {r+1,...,r+POS-1} . Up to this point the minimum of |Q|*BIG, QI[r+1],...,QI[r+POS-1] has been accumulated in MINVAL[r] . After HAVERESULT[r] is set to 1, MINVAL[r] is unchanged (because MINVAL[r] $\le$ BIGVECTOR[r] = |Q|*BIG) . Thus if SR(r,Q) = ∅ MINVAL[r] = |Q|*BIG and when this is decoded RESULT[r] = 0 . Otherwise MINVAL[r] = min(QI[r+1],..., QI[r+POS-1]) = (by definition) QI[R(r,Q)] . When this is decoded, RESULT[r] = R(r,Q) .

(b) Clearly each of the vectors used is O(|Q|) in length. Thus each iteration of step [2] takes O(|Q|) and a O(LIMIT*|Q|) time bound follows. ∎



We are still faced with the problem of choosing LIMIT so that if the value of R(r,Q) is needed HAVERESULT[r] = 1 . One way (quite obviously) is to set LIMIT = |Q| . Does this mean Alg 9's time bound is really $O(|Q|^2)$? In practice not. Recall that the input in general will be of the form $\#E_1\#E_2\#\ldots\#E_m\#$ . If the number of operators in each $E_r$ is < k then we may set LIMIT = k (since the R values of the operators cannot extend beyond the #'s). Since in practice individual expressions are fairly short, we can set LIMIT to some reasonably small fixed value (say 20) and a linear algorithm is obtained. Those very rare cases for which LIMIT is too small can be handled by Alg 7 and linearity is still maintained.

In conclusion then, we may utilize either Alg 7 or Alg 9 to compute the L and R functions. In either case the result is a compact, efficient and highly concurrent parser for arithmetic infix expressions.

## 5. Generating Object Code for AIG's

In compilation, parsing is not an end unto itself. Rather, it is closely connected with another phase of compilation - code generation. Since the actual code generated by a compiler depends on the particular computer for which it is targeted, we shall deal with a generalized machine instruction - the quadruple. A quadruple is of the form (OP,A,B,C) and has the semantics $C \leftarrow A$ OP $B$ .

It is quite easy to generate quadruples from the output of ALG 5. For each operator $\theta_j \neq \#$ we generate $(\theta_j, A, B, T_j)$ . $T_j$ is a temporary storage location associated uniquely with $j$ . A and B are determined as follows:

(1) If LEFT_SUBTREE[j] (RIGHT_SUBTREE[j]) is equal to 0 then A (B) is the first ID to $\theta_j$'s left (right).

(2) If LEFT_SUBTREE[j] (RIGHT_SUBTREE[j]) is equal to $r \neq 0$ then A (B) is $T_r$ .

Note that if $\theta_j$ is unary, A is null.

After the quadruples are generated they must be ordered correctly. One way to do this (recalling Thm 3) would be to sort the quadruples for each expression by the PREC values of the operators (in decending order). This approach is clumsy however, if only because it is not clear how to perform sorts efficiently on parallel computers. A better approach is as follows.

By Thm 4 (1), we know that $|SR(r,PREC(OPS))|$ = the size of $\theta_r$'s right subtree. Assume that RIGHT_SUBRREE_SIZE = $|SR(1,PREC(OPS))|,\ldots,|SR(|OPS|-1,PREC(OPS))|$ is calculated (it could be calculated when RIGHT_SUBTREE is). Since EXPR_ROOTS[r] ≡ RIGHT_SUBTREE[#_INDEX[r]] is the root of the r-th expression in INPUT, RIGHT_SUBTREE_SIZE[#_INDEX[r]] is the size of the r-th expression. We will generate quadruples in the following order:

{quadruples for $\theta_j$'s left subtree}
{quadruples for $\theta_j$'s right subtree}
{$\theta_j$'s quadruple}

Assume the quadruples for the r-th expression are to start at FIRST_ADR[r] .

The very last quadruple to be generated will be the one associated with the root of expression r's reduced derivation tree. This is j = EXPR_ROOT[r] . It's address (call it QUAD_ADR[j]) = FIRST_ADR[r] + RIGHT_SUBTREE_SIZE[#_INDEX[r]] - 1 . Once the address of $\theta_j$'s quadruple is known, the addresses of the other



quadruples in the r-th expression can be determined by iterating the following:

If the address of $\theta_k$'s quadruple is known (QUAD_ADR[k]) , then if RIGHT_SUBTREE[k] = m ≠ 0 then QUAD_ADR[m] = QUAD_ADR[k] - 1 . Further, if $\theta_k$ is binary and LEFT_SUBTREE[k] = p > 0 then QUAD_ADR[p] = QUAD_ADR[k] - RIGHT_SUBTREE_SIZE[k] - 1 .

For example, reconsidering the expression used in section 3,

$$\#ID_1*(ID_2+ID_3**ID_4) - SQRT\ ID_5\#SQRT((ID_6+ID_7)/ID_8)\# ,$$

it is easy to verify that

RIGHT_SUBTREE_SIZE = 5,2,1,0,1,0,3,2,0,0

If FIRST_ADR = L1,L2 , then by applying the above we get

QUAD_ADR = ?,L1+2,L1+1,L1,L1+4,L1+3,?,L2+2,L2,L2+1

where ? denotes undefined. We may use this to generate and order the following quadruples

| | | | |
|---|---|---|---|
| L1: | $(**,ID_3,ID_4,T_4)$ | L2: | $(+,ID_6,ID_7,T_9)$ |
| L1+1: | $(+,ID_2,T_4,T_3)$ | L2+1: | $(/,T_9,ID_8,T_{10})$ |
| L1+2: | $(*,ID_1,T_3,T_2)$ | L2+2: | $(SQRT, \ , T_{10},T_8)$ |
| L1+3: | $(SQRT, \ ,ID_5,T_6)$ | | |
| L1+4: | $(-,T_2,T_6,T_5)$ | | |

These quadruples correctly compute the above expressions. To see that the quadruples generated are always correct, simply note that:

(1) The quadruples to evaluate an operator's left and right subtrees are always generated <u>before</u> the operator's quadruple.

(2) If $\theta_i$ is some operator, then its quadruple stores its value in $T_i$ . Also, if $\theta_i$ roots the left (or right) subtree of $\theta_j$ , then $\theta_j$'s quadruple assumes its left (or right) argument to be in $T_i$ .

(3) Once a result is stored in a temporary, it is never changed since each temporary is assigned to <u>exactly</u> once.

Note that condition (3) points up a major weakness of our current method of allocating temporaries. Since each temporary is assigned to but once, we need as many temporaries as there are operators to evaluate an expression. Clearly a method of allocation which "reuses" temporaries is needed. One solution is as follows.

Assume that for each operator $\theta_j \neq \#$ we maintain a field RESULT_TEMP. If RESULT_TEMP[j] = k , then $\theta_j$'s quadruple stores its result into $T_k$ . For EXPR_ROOT[i] (the root of the i-th expression in INPUT) we set RESULT_TEMP[EXPR_ROOT[i]] = EXPR_ROOT[i] . The values of RESULT_TEMP for the other operators in the i-th expression are obtained by iterating the following:

> If RESULT_TEMP is known for $\theta_j$ then if $\theta_j$'s right subtree is rooted by $\theta_k$ then RESULT_TEMP[k] = RESULT_TEMP[j] . Further, if $\theta_j$ is binary and $\theta_j$'s left subtree is rooted by $\theta_p$ then if $\theta_j$'s right subtree is an ID then RESULT_TEMP[p] = RESULT_TEMP[j] else RESULT_TEMP[p] = p .

Note that the value of RESULT_TEMP is passed, if possible, to the right son of an operator. If the right son is an ID then it is passed to the left son (if it is not an ID). By using this allocation scheme in the above example we obtain:

RESULT_TEMP = ?,2,2,2,5,5,?,8,8,8 .

The corresponding quadruples are:

| | | | |
|---|---|---|---|
| L1: | $(**,ID_3,ID_4,T_2)$ | L2: | $(+,ID_6,ID_7,T_8)$ |
| L1+1: | $(+,ID_2,T_2,T_2)$ | L2+1: | $(/,T_8,ID_8,T_8)$ |
| L1+2: | $(*,ID_1,T_2,T_2)$ | L2+2: | $(SQRT, \;,T_8,T_8)$ |
| L1+3: | $(SQRT, \;,ID_5,T_5)$ | | |
| L1+4: | $(-,T_2,T_5,T_5)$ | | |

Note that 2 (rather than 5) and 1 (rather than 3) distinct temporaries are used. In general, it is easy to verify, that the number of different temporaries used is equal to the number of operators having <u>only</u> ID's as operands. While this is not always



the minimum number of temporaries required, it is usually a good approximation.

We are now ready to consider an algorithm which both parses and generates quadruples for arithmetic infix expressions. Let B be a boolean vector and V an arbitrary vector and assume |B| = |V| . Then B <u>COMPRESS</u> V is a vector composed of those elements of V whose corresponding B elements are 1. Thus 1,0,1,1 <u>COMPRESS</u> 5,6,7,8 = 5,7,8 . (Note that <u>COMPRESS</u> is an actual STAR-100 instruction).

<u>Algorithm 11</u>. (An AIG parser and code generator)

Input: (a) m arithmetic infix expressions, separated by #'s, stored in INPUT

(b) FIRST_ADR, a vector of length m equal to the starting addresses of the m sets of quadruples which will be generated.

Output: Encodings of quadruples to evaluate the m expressions of INPUT stored in QUAD_ADR, RESULT_TEMP, LEFT_SUBTREE and RIGHT_SUBTREE.

```
[1] Use Alg 5 to parse INPUT
[2] (Calculate QUAD_ADR and RESULT_TEMP as follows)
    RIGHT_SUBTREE_SIZE = |SR(1,PREC(OPS))|,...,|SR(|OPS|-1,
         PREC(OPS))|
    QUAD_ADR[EXPR_ROOTS] = FIRST_ADR +
         RIGHT_SUBTREE_SIZE[#_INDEX] - 1
    RESULT_TEMP = 1,...,|OPS| - 1
    OP_VECT = EXPR_ROOTS

    Do while  |OP_VECT| > 0
         RIGHT_SONS = RIGHT_SUBTREE[OP_VECT]
         LEFT_SONS = LEFT_SUBTREE[OP_VECT]
         QUAD_ADR[(RIGHT_SONS > 0) COMPRESS RIGHT_SONS] =
              QUAD_ADR[(RIGHT_SONS > 0) COMPRESS OP_VECT] - 1
```

QUAD_ADR[(LEFT_SONS > 0) COMPRESS LEFT_SONS] =
    QUAD_ADR[(LEFT_SONS > 0) COMPRESS OP_VECT] -
    RIGHT_SUBTREE_SIZE[(LEFT_SONS > 0) COMPRESS
        OP_VECT] - 1

LARGER_SONS = Max(LEFT_SONS,RIGHT_SONS)

RESULT_TEMP[(LARGER_SONS > 0) COMPRESS LARGER_SONS] =
    RESULT_TEMP[(LARGER_SONS > 0) COMPRESS OP_VECT]

BOTH_SONS = LEFT_SONS CONCAT RIGHT_SONS

OP_VECT = (BOTH_SONS > 0) COMPRESS BOTH_SONS ∎

The encoding of the quadruples is very similar to that used in Alg 5. Associated with each operator $\theta_j \neq \#$ is a quadruple. Its location is QUAD_ADR[j] . If RIGHT_SUBTREE[j] = 0 then the quadruples right argument is the first ID to the right of $\theta_j$ . If RIGHT_SUBTREE[j] $= k \neq 0$ then the right argument is $T_p$ where p = RESULT_TEMP[k] . Similarly, if LEFT_SUBTREE[j] = 0 then the quadruple's left argument is the first ID to $\theta_j$'s left and if LEFT_SUBTREE[j] = k > 0 then the argument is $T_p$ where again p = RESULT_TEMP[k] . Further, if LEFT_SUBTREE[j] = - 1 then $\theta_j$ is unary and thus has no left argument. The quadruple stores its result in RESULT_TEMP[j] .

Theorem 12. Let G be some AIG. Then

(1) If INPUT $\varepsilon$ L(G) and INPUT contains m arithmetic infix expressions then
   (a) Alg 11 will produce encodings of m sets of quadruples. The i-th set of quadruples $(1 \le i \le m)$ will correctly evaluate the i-th arithmetic expression and will start at FIRST_ADR[i] .
   (b) If the L and R functions and RIGHT_SUBTREE_SIZE are evaluated in O(|INPUT|) time and space then Alg 11 will execute in O(|INPUT|) time and space.

(2) If INPUT $\notin$ L(G) then Alg 11 will signal an error and stop.



Proof: (1a) By Thm 6(1), Alg 5 parses INPUT correctly and by Thm 4(1) RIGHT_SUBTREE_SIZE contains correct values. We first establish the following 3 facts:

(i) Let $\theta_j \neq \#$ root a subtree containing k operators. Then the k quadruples associated with these operators have addresses QUAD_ADR[j] + 1 - k,...,QUAD_ADR[j] . (This can be established via a simple induction on the height of $\theta_j$'s subtree)

(ii) Let $\theta_j$ be as in (i). Then if $\theta_q$ is in the subtree rooted by $\theta_j$ then RESULT_TEMP[q] = RESULT_TEMP[j] or = p where $\theta_p$ is in $\theta_j$'s subtree. (Again a simple induction on the height of $\theta_j$'s subtree establishes this)

(iii) Let $\theta_r$ be any operator $\neq \#$ . Then RESULT_TEMP[r] = s where $\theta_s$ roots a subtree containing $\theta_r$ . (Initially, if EXPR_ROOTS[i] = t for $1 \le i \le m$ then RESULT_TEMP[t] = t . Further, if $\theta_u$ is a son of $\theta_v$ then RESULT_TEMP[u] = u or RESULT_TEMP[u] = RESULT_TEMP[v]).

Let $\theta_j$ be as in (i) and assume QUAD_ADR[j] = r . Then we will establish that if the quadruples at r-k+1,...,r are executed then the value of the subtree rooted by $\theta_j$ will be computed correctly and stored in RESULT_TEMP[j] .

Consider first the case in which $\theta_j$'s subtree is of height 1. Then all of $\theta_j$'s arguments are ID's and the single quadruple at r correctly computes the expression. By induction, we now assume the above to be established for subtrees of height $\le m$ and consider subtrees of height m + 1 . If $\theta_j$ has only one son, $\theta_p$ , that is an operator (the other son is an ID or $\theta_j$ is unary), it must be of height m and it contains k - 1 operators. But then quadruples r-k+1,...,r-1 correctly compute $\theta_p$'s subtree and store the result in RESULT_QUAD[p] . $\theta_j$'s quadruple at r immediately uses the result (and perhaps an ID) to correctly compute $\theta_j$'s subtree. If $\theta_j$ has two sons $\theta_p$ and $\theta_q$ which are operators then the height of each subtree is $\le$ m . Further, if $\theta_q$'s subtree has t operators, then $\theta_p$'s subtree has k - t - 1 . This means QUAD_ADR[q] = r - 1 and QUAD_ADR[p] = r - 1 - t .

Thus executing quadruples $r - k + 1$ to $r - 1 - t$ correctly evaluates $\theta_p$'s subtree and stores the result in RESULT_TEMP[p] and executing quadruples $r - t$ to $r - 1$ correctly evaluates $\theta_q$'s subtree and stores the result in RESULT_TEMP[q] . Now by construction RESULT_TEMP[q] = RESULT_TEMP[j] and RESULT_TEMP[p] = p ǂ RESULT_TEMP[q] (by (iii)). Further, $\theta_p$ <u>does not occur</u> in $\theta_q$'s subtree. Therefore by (ii), no operator in $\theta_q$'s subtree has RESULT_TEMP = p . Thus if quadruples $r - k + 1$ to $r - 1$ are executed, RESULT_TEMP[p] and RESULT_TEMP[q] contain the values of $\theta_j$'s left and right subtrees. Thus executing quadruples $r - k + 1$ to $r$ correctly evaluates $\theta_j$'s subtree.

Letting $\theta_j$ = the root of the i-th expression yields the desired result.

(1b) By Thm 6(1) step [1] is linear. By assumption the calculation of RIGHT_SUBTREE_SIZE also is. All vectors used are bounded by $|OPS| < |INPUT|$ . Further each iteration of the while loop is proportional to $|OP\_VECT|$ . But each operator ǂ # in OPS appears in OP_VECT exactly once.

(2) Follows from Thm 6(2). ∎

Note that RIGHT_SUBTREE SIZE can easily be calculated when RIGHT_SUBTREE is. In fact, in Alg 7 when RIGHT_SUBTREE[i] = RESULT[i] = R(i,Q) is stored, RIGHT_SUBTREE_SIZE[i] = |SR(i,Q)| = POINTER - i - 1 . In like manner, when Alg 9 sets HAVERESULT[i] = 1 , |SR(i,Q)| = POS - 1 . Thus the results of Thms 8 and 10 immediately apply to the calculation of RIGHT_SUBTREE_SIZE as well as RIGHT_SUBTREE. Finally, it is important to observe that the effect of the while loop in Alg 11 is to examine the roots of <u>all</u> the individual expressions in INPUT, then <u>all</u> their sons, etc. This means that the while loop, as well as the rest of the algorithm, can be executed in a highly concurrent manner.



## 6. Conclusion

As we have seen, it is possible to develop compact, efficient and highly concurrent algorithms to parse and generate code for arithmetic expressions. The question of how effective such algorithms would be in actual parallel environments remains unanswered. Clearly, careful implementations and tests are called for. However preliminary tests are favorable.

APL may be used to model parallel vector computers such as the STAR-100. This is because in APL it is much faster to perform an operation on a vector than it is to perform the same operation iteratively on its components.

A slight variant of Alg 11 (using Alg 9 to compute the L and R functions) was coded in APL\360 as was a conventional serial SLR(1) parser and code generator. In all cases in which INPUT contained more than a minimum number of operators (5 to 7), Alg. 11 proved faster. As the number of operators in INPUT was increased, a limiting ratio of approximately 4 to 1 in execution times was quickly reached.

These tests, of course, are by no means conclusive but they do suggest that the above algorithms could be most competitive with serial techniques on suitable parallel computers.

## References


1. Fischer, C. "On Parsing Context Free Languages in Parallel Environments". Ph.D. Thesis, Cornell University, Ithaca, N.Y., 1975.

2. Floyd, R. "Syntactic Analysis and Operator Precedence". JACM, 10:3, pp. 316-333, 1963.

3. Lincoln, N. "Parallel Programming Techniques for Compilers". SIGPLAN Notices, Vol. 5, No. 10, 1970.

4. Zozel, M. "A Parallel Approach to Compilation". ACM Symposium on the Principles of Programming Languages, pp. 59-70, 1973.