Parallel Parsing of Languages Generated by Ambiguous Bounded Context Grammars

Robert McCloskey
University of Scranton
mccloske@cs.uofs.edu

Jie Wang [*]
University of North Carolina at Greensboro
wangj@hamlet.uncg.edu

Jay Belanger
Northeast Missouri State University

Abstract

Using the CRCW PRAM model, we describe a language recognition algorithm for an arbitrary grammar in the class of BCPP grammars [9]. (BCPP grammars, which admit ambiguity, are a generalization of both the NTS grammars [14] and Floyd's bounded context (BC) grammars [4].) Using n processors, the algorithm runs in time $O(h \cdot \log n)$ ($O(h)$ in the case of an unambiguous grammar), where n is the length of the input string x and h is bounded above by the height of the tallest parse tree for x. Estimating that $h \in O(\log n)$ in the usual case, this yields a running time of $O(\log^2 n)$ ($O(\log n)$ in the unambiguous case).

Keywords: ambiguous grammar, parallel algorithm, parsing

1 Introduction

Ever since it was discovered, around 1960, that a context-free grammar (CFG) was capable of describing, to a close approximation, the syntax of a typical high-level programming language, the problem of parsing (with respect to such a grammar) has been of great interest among computer scientists, including both theoreticians and practitioners. As a consequence of all the attention this problem has received, a large number of interesting and useful concepts have arisen, including the identification of several subclasses of CFG's (e.g., LL(k), LR(k), LALR(k), precedence, bounded right context, etc. [2]) not only suitable for modeling the syntactic features of programming languages, but also having the property that, given any grammar in the class, an efficient (i.e., linear time) parsing algorithm can be constructed automatically. The practical results have included tools such as compiler compilers and parser generators (e.g., yacc) and, more generally, a coherent and applicable theory for the lexical and syntactic analysis phases of compiling.

[*]Supported in part by the NSF under grant CCR-9108899.

In anticipation of the widespread use of parallel computers, in recent years there has been much work in designing algorithms suitable for implementation on such machines. By now, parallel algorithms for virtually every classic problem in computer science have appeared in the literature, including parsing [3, 7, 8, 11, 12, 13, 15].

However, most work on parsing has focused on either the entire class of CFG's or else on subclasses (such as those listed earlier) that

(1) can be parsed efficiently via a left-to-right scan of the input string, and

(2) do not admit ambiguity.

An exception to (2) is [1], in which "disambiguating rules" are used for augmenting an LR or LL grammar in order to resolve conflicts in its parse table. (For some grammars, however, modifying the parse table in this way results in the parser recognizing only a proper subset of the language described by the grammar.) With respect to (1), it is natural to focus on grammars having this property if one is interested in only sequential parsing algorithms, such as was the case in the 1960's. The result of "parallelizing" such parsers can be kind of messy, however, due to their inherent reliance on left-to-right scanning. (See the algorithms described in [3, 8], for example.)

In this paper, we present a parallel language recognition algorithm for an arbitrary grammar in the class of BCPP ("bounded context parsing property") grammars. This class properly includes the NTS grammars [14] as well as Floyd's bounded context (BC) grammars [4], and, of particular interest to us, includes grammars that are ambiguous. BCPP languages possess several interesting language-theoretic properties, and one surmises that the BCPP concept is a closer approximation to "human parsability" than is provided by the more powerful deterministic pushdown automaton model [9].

Although BCPP languages are properly included within the deterministic context-free languages (DCFLs), they would seem to include all the (context-free) features of programming language syntax. Furthermore, a BCPP grammar, being naturally suited for the task, gives rise to a straightforward and efficient parallel parsing algorithm using a linear number of processors. This is in contrast to parallel parsing algorithms that (1) are applicable to a wider class of languages (such as the CFLs or the DCFLs), but which use a quadratic (or greater) number of processors [7], or (2) are based on LR(k) grammars, but which tend to be rather complicated and for which upper bounds on running time are linear [3, 8], or (3) are applicable to classes of grammars or languages properly included in BCPP [11, 12].

We note that for LL grammars there is a parsing algorithm that runs in logarithmic time on a linear number of processors [15]. Although this algorithm competes well with ours, it is not applicable to ambiguous grammars, and thus cannot be viewed as generalizing the result reported here.

```
     E                   E
    / \                 /|\
   -   E               E + E
      /|\             /|   |
     E + E           - E   a
     |   |             |
     a   a             a
```

Figure 1: Two Parse Trees for $-a+a$

2 Review of Standard Terminology

An *alphabet* Γ is a finite set of symbols. A *string* η over Γ is a sequence of symbols from Γ. The length of a string η is denoted by $|\eta|$. A *language* over Γ is a set of strings over Γ. The language containing all such strings (respectively, all strings of length k) is denoted by Γ^* (respectively, Γ^k).

A *context-free grammar* is a quadruple $G = (N, \Sigma, P, S)$, where N and Σ are disjoint alphabets of *nonterminal* and *terminal* symbols, respectively, $S \in N$ is the *start symbol*, and $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *productions*. A production $(Q, \beta) \in P$ is usually written $Q \to \beta$, and we call Q its *left-hand side* and β its *right-hand side*. We let $V = N \cup \Sigma$, and by convention we use upper case letters (e.g., E, Q, S) to denote nonterminals, lower case letters (e.g., a) to denote terminals, x to denote a string in Σ^*, and α, β, γ, η, ϕ, and ψ to denote strings in V^*.

Strings ϕ and ψ stand in relation $\phi \Rightarrow \psi$ if there exist $Q \in N$ and strings α, β, and γ such that $\phi = \alpha Q \gamma$, $\psi = \alpha\beta\gamma$, and $Q \to \beta \in P$. The reflexive, transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$ and is of central importance: $\phi \overset{*}{\Rightarrow} \psi$ means that there exist strings $\phi_0, \phi_1, \ldots, \phi_n$, $n \geq 0$, such that $\phi = \phi_0 \Rightarrow \phi_1 \Rightarrow \cdots \Rightarrow \phi_n = \psi$. Such a sequence of steps is called a *derivation* of ψ from ϕ.

A string η is said to be a *sentential form* of G if $S \overset{*}{\Rightarrow} \eta$. The set of sentential forms of G, i.e., $\{\eta \in V^* \mid S \overset{*}{\Rightarrow} \eta\}$, is denoted by $\hat{L}(G)$. The *language of* G, denoted $L(G)$, consists of all sentential forms comprised of only terminal symbols, i.e., the set $\hat{L}(G) \cap \Sigma^*$.

For every sentential form η of a grammar, there is at least one parse tree whose leaves, when read from left to right, spell out η. (The root of the tree is labeled S and the branches correspond in the obvious way to applications of the productions.) A grammar is said to be *unambiguous* if every sentential form has a unique parse tree. The grammar G_1 (which you will easily recognize as one that generates arithmetic expressions involving binary $+$ and unary $-$) with productions

$$E \;\to\; E + E \;\mid\; -E \;\mid\; a$$

is ambiguous, as can be seen from Figure 1, which shows distinct parse trees for the string $-a+a$.

If $S \overset{*}{\Rightarrow} \phi Q \psi \Rightarrow \phi\beta\psi$, the noted occurrence of β is said to be a (Q-)*phrase*, as its appearance is due to an application of the production $Q \to \beta$. Clearly, this is equivalent to there existing a parse tree for $\phi\beta\psi$ in which the leaves spelling out β are connected above to a parent labeled Q. The process of attempting to construct a parse tree or derivation (from S) for a given string

$x \in \Sigma^*$ is called *parsing*. Here, we are interested in *bottom-up parsing*, in which the steps of a derivation are determined in reverse order, starting with x and ending, if possible, with S. To do this, we identify one or more (mutually nonoverlapping) phrases in the current string, *reduce* each of them to the corresponding nonterminal symbol, and then repeat until the current string is such that no further reductions are possible. (This is analogous to constructing the parse tree from the leaves upward, where at each step we draw edges upward from the nodes spelling out a phrase to a new parent node labeled by the appropriate nonterminal.)

Note that the mere occurrence of a string β in a sentential form $\phi\beta\psi$, where $Q \to \beta \in P$, does not imply its being a phrase, as it may be that $\phi Q\psi$ is *not* a sentential form, in which case it would be a "mistake" to reduce β to Q. The significance of grammar classes such as LR(k) and BC is precisely that they provide easy-to-check conditions that prevent us from making such mistakes during parsing.

3 BCPP Grammars

Intuitively, a BCPP grammar is one such that, given a sentential form $\phi\beta\psi$, one knows whether or not it is correct to reduce the noted occurrence of β to Q by examining only a fixed ("bounded") number of symbols occurring to the left and to the right of β. In what follows, we formalize this notion; the grammar $G = (N, \Sigma, P, S)$ is to be understood throughout.

Henceforth, we use V to denote the alphabet of G augmented with two new symbols \$ and # that occur in neither N nor Σ, i.e., let $V = N \cup \Sigma \cup \{\$, \#\}$. We will assume that each sentential form is padded on the left by the string $\$^m$ and on the right by the string $\#^n$ for some suitably chosen nonnegative integers m and n.

In a string $\phi\alpha\beta\gamma\psi$, where $|\alpha| = m$ and $|\gamma| = n$, the noted instance of β is said to occur in the (m,n)-*context* (α,γ). The set of (m,n)-contexts in which a given nonterminal symbol can appear is of interest to us, and so we state the following definition.

Definition 1 *For $Q \in N$ and $m, n \geq 0$,*

$$\mathtt{Ctx}_{m,n}(Q) = \{(\alpha,\gamma) \in V^m \times V^n \mid \exists \phi,\psi : \$^m S \#^n \overset{*}{\Rightarrow} \phi\alpha Q\gamma\psi \}.$$

With this, we can formally define the BCPP concept.

Definition 2 *A production $Q \to \beta$ is said to be (m,n)-BCPP if for every $\phi, \psi \in V^*$, $\alpha \in V^m$, and $\gamma \in V^n$, the two conditions*

$$\$^m S \#^n \overset{*}{\Rightarrow} \phi\alpha\beta\gamma\psi \quad \text{and} \quad (\alpha,\gamma) \in \mathtt{Ctx}_{m,n}(Q)$$

together imply $\$^m S \#^n \overset{}{\Rightarrow} \phi\alpha Q\gamma\psi$.*

In other words, $Q \to \beta$ is (m,n)-BCPP if any occurrence of β with α to its left and γ to its right, where $(\alpha,\gamma) \in \mathtt{Ctx}_{m,n}(Q)$, can be (correctly) reduced to Q.

From the definition, it is clear that if a production is (m,n)-BCPP, then it is also (m',n')-BCPP for any m' and n' satisfying $m' \geq m$ and $n' \geq n$. We will feel free to use this fact without mentioning it again.

Definition 3 *A grammar G is said to be (m,n)-BCPP if every production in G is (m,n)-BCPP.*

By further stipulating that the grammar be unambiguous, we describe the (m,n)-BC grammars of [4]. We note, in passing, that the problem of determining whether G is (m,n)-BCPP, for given G, m, and n, is decidable [9], although it seems likely that no efficient algorithm exists (as is the case with LR(k) grammars [5]).

As an example, consider the grammar G_2 having the productions

$$(1)(2)(3)(4)(5) \quad E \quad \rightarrow \quad E+E \mid E-E \mid \ -E \mid a \mid aa$$

(This extends the grammar G_1 presented earlier.) It is not difficult to verify that

$$\mathtt{Ctx}_{1,1}(E) = \{\$,+,-\} \times \{\#,+,-\}$$

We shall not formally demonstrate that G_2 is $(1,1)$-BCPP; rather, we simply state the following facts, from which this result follows. Productions (1), (2), and (5) are $(0,0)$-BCPP because any occurrence of $E+E$ or $E-E$ or aa is correctly reduced to E, regardless of the context in which it occurs. Production (3) is $(1,0)$-BCPP because any occurrence of $-E$ preceded by \$, $+$, or $-$ is correctly reduced to E. (It is *not* $(0,0)$-BCPP, however, because reducing $-E$ to E in the sentential form $E-E$, for example, is incorrect, because it yields EE, which is not a sentential form.) Production (4) is $(1,1)$-BCPP because any occurrence of a preceded by \$, $+$, or $-$ and followed by #, $+$, or $-$ is correctly reduced to E. (It is neither $(0,1)$-BCPP nor $(1,0)$-BCPP, however, because reducing either of the a's in the (padded) sentential form $\$aa\#$, for example, is incorrect.)

Parsing based on an ambiguous grammar presents us with the problem of having to resolve conflicts between overlapping phrases. For example, in parsing $-E+E$ (see Figure 1), we must choose either to reduce $E+E$ to E (according to Production (1)) or to reduce $-E$ to E (according to Production (3)); we cannot do both. This problem is magnified in a parallel setting, where its resolution may require that distinct processors cooperate with each other.

4 BCPP Parsing Algorithm

Our model of computation is the CRCW PRAM (Concurrent Read, Concurrent Write Parallel Random Access Machine), which, although not being entirely realistic, serves as a useful means by which to study the nature of parallel computation [6, 10].

A PRAM is a collection of synchronous processors (each one being a sequential RAM with local memory) executing in parallel and communicating through a shared, global memory. In a single unit of time, each processor can read a (global or local) memory location, execute a RAM instruction, and write to a (global or local) memory location. “Concurrent” read (resp., write) means that any global memory location can be read (resp., written), in a single unit of time, by an arbitrary number of processors. We adopt the so-called COMMON model for resolving write conflicts, which says that any two processors writing to the same location at the same time *must* write the same value.

The input to our algorithm is a string x that is to be tested for membership in $L(G)$, where G is a fixed (m, n)-BCPP grammar. We assume, without loss of generality, that $n > 0$ and that G has no productions of the form $Q \to \lambda$, where λ is the empty string [9]. We let ℓ denote the length of the longest right-hand side of any production in G.

The central idea behind the algorithm is that, in parallel, we can locate, and then reduce, a maximal set of mutually nonoverlapping phrases occurring in the current string. From the point of view of building a parse tree, each such step reduces, by one, the height of that portion of the tree remaining to be constructed. Thus, the number of steps required to complete a parse of x is bounded above by the height h of the tallest parse tree for any substring of x, where now we are including trees whose root is not necessarily labeled by the start symbol. (In the case that $x \in L(G)$, the tallest such tree is a parse tree for x itself.) It is not possible to express h precisely as a function of $|x|$, because h depends not only on the length of x but on x itself, as well as G. At one extreme, G could be a linear grammar (i.e., one in which at most one nonterminal appears on the right-hand side of any production), in which case h is linear in $|x|$. At the other extreme, G could be such that the right-hand side of every production has either zero or at least two nonterminals, in which case $h \leq \lfloor \log_2 |x| \rfloor + 1$. One suspects that, for "most" grammars, $h \in \mathrm{O}(\log |x|)$. Also, there is a simple method for modifying a BCPP grammar that tends to increase the number of nonterminals occurring in the right-hand sides of its productions, thereby tending to decrease the heights of its parse trees [9].

The idea of parsing by identifying and reducing as many phrases as possible, in parallel, is not new; what is interesting is that, because our parsing is based upon a BCPP grammar, the resulting algorithm is very straightforward and requires only a linear number of processors to run efficiently. The solution to the problem introduced by ambiguity—that of resolving conflicts between overlapping phrases—is also interesting.

```
FindPhrases;
while PhrasesExist do
    ResolveConflicts;
    PerformReductions;
    FindPhrases;
if final string is $\$^m S \#^n$
    then ACCEPT;
    else REJECT;
```

Figure 2: BCPP Parallel Parsing Algorithm

The high-level description of the algorithm is in Figure 2 and is self-explanatory. To present the algorithm in more detail, we now describe the assumed configuration of the input string, the supporting data structures, and each of the algorithm's components. Figures 3 through 6 give detailed listings. (An example of parsing according to the algorithm is given at the end of this section.)

We assume that the (padded) input string $\$^m x \#^n = a_1 a_2 \cdots a_t$ is stored as a linked list represented by the global arrays `Next[1..t]` and `Symbol[1..t]`. More precisely, the index h of the head node (which need not be known) is such that $\texttt{Symbol}[\texttt{Next}^i[h]] = a_{i+1}$ for all i, $0 \leq i < t$, where we inductively define $\texttt{Next}^0[\texttt{k}] = \texttt{k}$ and $\texttt{Next}^{\texttt{j}+1}[\texttt{k}] = \texttt{Next}[\texttt{Next}^{\texttt{j}}[\texttt{k}]]$. The index r of the rear node (which corresponds to the rightmost $\#$ endmarker) is such that `Next[r] = r`, i.e., the rear points to itself. The remainder of the shared variables used in the program are as follows:

`Active[i]`: (boolean) tells whether node i is still active.
`Symbol[i]`: symbol stored in node i.
`Next[i]`: successor of node i in the list.
`TempNext[i]`: auxiliary array used in the `ListRank` subprogram.
`Rank[i]`: distance from node i to end of list (node r).
`LocRank[i]`: equals `Rank[i] mod` ℓ; used in resolving reduce conflicts.
`Reduce[i]`: (boolean) indicates whether to reduce a phrase beginning at node i.
`ReduceBy[i]`: indicates by which production (if any) to reduce the phrase beginning at node i.
`PhrasesExist`: (boolean) indicates whether current string contains any phrases.

Now we describe the components of the algorithm:

`FindPhrases`: For each active node i, the processor assigned to it traverses the sublist of length $m + \ell + n$ beginning there, and, in so doing, constructs the string η which is the concatenation of the symbols stored in the nodes of that sublist. The processor then determines whether the $(m+1)$-st symbol of η is the beginning of a phrase, i.e., whether there exist strings α, β, γ, and ϕ and a nonterminal Q such that $\eta = \alpha\beta\gamma\phi$, $Q \to \beta$ is a production in G, and $(\alpha, \gamma) \in \texttt{Ctx}_{m,n}(Q)$. (In an implementation, a processor would use some pattern matching device, such as a finite state machine, to help make this determination.) If the answer is yes, the processor sets $\texttt{Reduce}[\texttt{Next}^{\texttt{m}}[\texttt{i}]]$ to **true** and $\texttt{ReduceBy}[\texttt{Next}^{\texttt{m}}[\texttt{i}]]$ to $Q \to \beta$, indicating that there is a phrase (namely, β) contained in the sublist beginning at node $\texttt{Next}^{\texttt{m}}[\texttt{i}]$ and that it can be reduced to Q. *Running time*: $O(m + n + \ell)$, which is $O(1)$.

`ListRank`: Using the standard algorithm [10], we compute, for each active node i, the number of nodes that follow i in the list. *Running time*: $O(\log |x|)$.

`ResolveConflicts`: Two nodes are said to be in conflict if phrases begin at both of them and these phrases overlap. The strategy we choose to resolve such conflicts is as follows. First, compute each active node's rank (i.e., its distance from the rear of the list) using `ListRank`. Then, for each such node, compute its "local rank," which is given by its rank modulo ℓ.

We observe that in any contiguous sublist of length ℓ, the nodes have local ranks of, in order, j, $j-1$, …, 1, 0, $\ell - 1$, $\ell - 2$, …, $j + 2$, $j + 1$, for some j satisfying $0 \leq j < \ell$. By definition of ℓ, the ranks of two nodes in conflict must differ by less than ℓ; it follows that their local ranks are distinct.

```
FindPhrases:
    for each $i$ such that Active[i] in parallel do
      PhrasesExist := false;  k := $\mathtt{Next}^{\mathtt{m}}$[i];
      $\eta$ := $\prod_{0 \le j < m+n+\ell}$ Symbol[$\mathtt{Next}^{\mathtt{j}}$[i]];
      if $\exists \alpha, \beta, \gamma, \phi \in V^*$, $Q \in N$
           such that $Q \to \beta \in P \;\wedge\; (\alpha, \gamma) \in \mathtt{Ctx}_{m,n}(Q) \;\wedge\; \eta = \alpha\beta\gamma\phi$  then
        PhrasesExist := true;
        Reduce[k] := true;  ReducibleBy[k] := $Q \to \beta$;
      else
        Reduce[k] := false;
```

Figure 3: Finding Phrases

```
ListRank:
    for each $i$ such that Active[i] in parallel do
      if Next[i] = i
        then Rank[i] := 0;
        else Rank[i] := 1;
      TempNext[i] := Next[i];
      repeat $\lceil \log |x| \rceil$ times do
        Rank[i] := Rank[i] + Rank[TempNext[i]];
        TempNext[i] := TempNext[TempNext[i]];
```

Figure 4: List Ranking

The larger a node's local rank, the higher its "reduce priority". For any two conflicting nodes, the one with the higher priority will "kill" the other one (unless the first had already been killed by a node with still higher priority). By letting k vary from $\ell - 1$ down to 1 and having any surviving node of local rank k kill off all those nodes of smaller local rank with which it conflicts, we guarantee that the nodes surviving in the end will correspond to a maximal set of nonoverlapping phrases. (This yields to an inductive proof, which is omitted.)

Running time: This is dominated by the list ranking step, the running time of which is logarithmic in the length of the list. Since the length of the list is bounded above by $m + n + |x|$ (that being its initial length), we get a running time of $O(\log |x|)$. The running time of the rest is proportional to ℓ^2, which is a constant. (By using an approach that is more clever than what is shown in the code of Figure 5, we can lower this constant to one proportional to ℓ.)

```
ResolveConflicts:
    ListRank;
    for each $i$ such that Active[i] in parallel do
        LocRank[i] := Rank[i] mod $\ell$;
        for k := $\ell$ - 1 downto 1 do
            if LocRank[i] = k $\wedge$ Reduce[i] then
                Let $Q \to \beta$ be ReduceBy[i];
                (* traverse the sublist beginning at node i and
                (* kill all conflicting nodes of lower priority *)
                j := i;
                repeat min{ k, $|\beta| - 1$} times do
                    j := Next[j];
                    Reduce[j] := false; (* i kills j *)
            elseif LocRank[i] < k then
                (* beginning at node i, find the first node with local rank $k$,
                (* and allow self to be killed if that is a conflicting node *)
                j := i;
                repeat min{$\ell - 1, |\beta| - 1$} times do
                    j := Next[j];
                    if LocRank[j] = k $\wedge$ Reduce[j] then
                        Reduce[i] := false; (* j kills i *)
```

Figure 5: Resolving Conflicts

`PerformReductions`: For each node i such that `Reduce[i]` holds, reduce by the production $Q \to \beta$ specified by `ReduceBy[i]`. This entails writing the symbol 'Q' into node i, deactivating the next $|\beta| - 1$ nodes, and connecting node i to the one that follows. *Running time*: $O(\ell)$, which is $O(1)$.

Taking all the components into account, we get that each iteration of the loop in the main algorithm runs in time $O(\log |x|)$. (The time is dominated by the list ranking operation in `ResolveConflicts`.) Having already established that the number of such iterations is $O(h)$,

```
PerformReductions:
    for each $i$ such that Reduce[i] in parallel do
      Let $Q \to \beta$ be ReduceBy[i];
      Symbol[i] := 'Q'; (* reduce *)
      k := Next[i];
      repeat $|\beta| - 1$ times do
        Active[k] := false; (* deactivate rest of nodes *)
        k := Next[k];       (* holding reduced phrase *)
      Next[i] := k; (* update pointer to next active node *)
```

Figure 6: Performing Reductions

we get a total running time of $\mathrm{O}(h \cdot \log |x|)$, as claimed. If G is unambiguous, there is no need to resolve conflicts between overlapping phrases, because no two phrases can ever overlap. (If two appear to overlap, which can be checked in constant time, the parse is doomed to failure, and thus the algorithm can halt immediately and reject the input string.) Thus, we can discard the `ResolveConflict` operation from the loop, and thereby achieve time $\mathrm{O}(h)$.

An example is in order. Referring back to G_2, suppose that during the middle of a parse, we had arrived at the string $\$ - E + E - -E + E + -E\#$. Using subscripts to indicate the ranks of nodes at which phrases begin, we write this as

$$\$ -_{12} E_{11} + E - -_7 E_6 + E + -_2 E\#$$

Local ranks of nodes where phrases begin are as follows:

$$2 : -_2,\ E_{11} \qquad 1 :\ -_7 \qquad 0 : E_6,\ -_{12}$$

(Recall that a node's local rank is its rank modulo ℓ, and for G_2 we have $\ell = 3$.) In resolving conflicts, $-_2$ will kill no one, but E_{11} will kill $-_{12}$. Then $-_7$ will kill E_6. The phrases beginning at each of the surviving nodes ($-_2$, E_{11}, and $-_7$) will then be reduced, yielding a list containing the string $\$ - E - E + E + E\#$. Inserting subscripts as above (corresponding to the step of finding phrases in the new string), we get the string

$$\$ -_8 E_7 - E_5 + E_3 + E\#$$

Local ranks of nodes where phrases begin are as follows:

$$2 : E_5,\ -_8 \qquad 1 : E_7 \qquad 0 : E_3$$

To resolve the conflicts, E_5 kills E_3 and E_7, while $-_8$ kills E_7. Reductions at E_5 and $-_8$ yield $\$E - E + E\#$. After locating phrases, we get

$$\$E_5 - E_3 + E\#$$

E_5 has local rank two, and so it will kill the conflicting E_3, whose local rank is zero. Reduction yields the string $\$E + E\#$. This string has only one phrase; reducing it yields $\$E\#$, and we are done. The parse tree corresponding to this sequence of steps is shown in Figure 7.

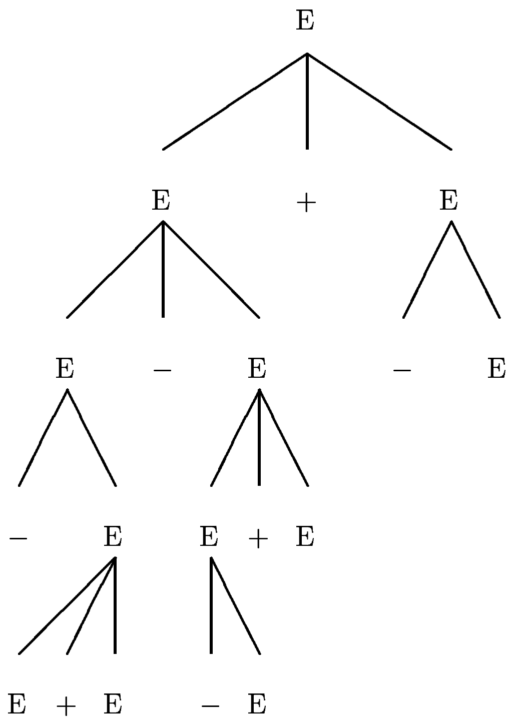

Figure 7: Parse tree for $-E+E--E+E+-E$

5 Conclusion

We have described an algorithm (on the CRCW PRAM model) that, for an arbitrary (fixed) (m, n)-BCPP grammar G, determines whether a given string x is in $L(G)$. The algorithm runs in time $\mathrm{O}(h \cdot \log |x|)$ using $|x|$ processors, where h is the height of the tallest parse tree for (any substring of) x. (In the case that G is known to be unambiguous (i.e., it is a BC grammar), the algorithm can be modified to run in time $\mathrm{O}(h)$.) This combination of running time and processor count compares well to other algorithms in the literature. But the main contribution of the paper is that it addresses the issue of parallel parsing with respect to ambiguous grammars, and in particular the problem of dealing with overlapping phrases. (We note that, although the grammars considered in [11] include ambiguous ones, they are such that no two phrases can overlap.)

Because it does not include the details for constructing the parse tree for x, our algorithm is, strictly speaking, a language recognition algorithm rather than a parsing algorithm. However, the reader can convince herself that it is possible to augment the algorithm to perform the task of constructing the parse tree, and to do so without increasing its asymptotic complexity.

References

[1] Aho, A.V., Johnson, S.C., and Ullman, J.D., Deterministic Parsing of Ambiguous Grammars, *Comm. ACM* 18:8 (1975), 441-452.

[2] Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing.* Prentice-Hall, Englewood Cliffs, N.J., 1972. *Vol. 2: Compilers*, 1973.

[3] Fischer, C., On Parsing Context-Free Languages in Parallel Environments, *Ph.D. Thesis*, Cornell University, Ithaca, NY (1975).

[4] Floyd, R.W., Bounded Context Syntactic Analysis, *Comm. ACM* 7:2 (1964), 62-67.

[5] Hunt, H.B., Szymanski, T.G., and Ullman, J.D., On the Complexity of LR(k) Testing, *Comm. ACM* 18:12 (1975), 707-716.

[6] Karp, R.M. and Ramachandran, V., Parallel Algorithms for Shared-Memory Machines, in: Van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, MIT/Elsevier (1990), 869-941.

[7] Klein, P. and Reif, J., Parallel Time O($\log n$) Acceptance of Deterministic CFLs on an Exclusive-Write P-RAM, *SIAM Journal on Computing*, 17:3 (1988), 463-485.

[8] Ligett, D., McCluskey, G., and McKeeman, W. M., Parallel LR Parsing, *Tech. Report TR-82-03*, Wang Institute of Graduate Studies, School of Information Technology, Tyngsboro, MA (1982).

[9] McCloskey, R., Bounded Context Grammars and Languages, *Ph.D. Thesis*, Renssselaer Polytechnic Institute, Troy, NY (1992).

[10] Reif, J. (ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993.

[11] Rytter, W. and Giancarlo, R., Optimal Parallel Parsing of Bracket Languages, in: Albrecht, A., Jung, H., and Mehlhorn, K., eds., *Parallel Algorithms and Architectures*, Lect. Notes in Comp. Sci. 269, (1987), Springer-Verlag, 146-154.

[12] Sarkar, D. and Deo, N., An Optimal Parsing Algorithm for a Class of Block-Structured Languages, in: Sahni, S., ed., *Proc. 1987 Int'l Conf. on Parallel Processing*, (1987), 585-588.

[13] Sarkar, D. and Deo, N., Estimating the Speedup in Parallel Parsing, *IEEE Trans. on Software Engineering*, 16:7 (1990), 677-683.

[14] Senizergues, G., The Equivalence and Inclusion Problems for NTS Languages, *J. Comput. Sys. Sci.* 31 (1985), 303-331.

[15] Skillicorn, D.B. and Barnard, D.T., Parallel Parsing on the Connection Machine, *Info. Proc. Let.*, 31 (1989), 111-117.