A Parallel Version of Earley's Algorithm[*]

Danilo Bruschi Giovanni Pighizzini

Dipartimento di Scienze dell'Informazione
Università degli Studi
Via Comelico 39, 20135 Milano – ITALY
`{bruschi,pighizzi}@imiucca.csi.unimi.it`

TECH. REP. 114–94

Abstract

In the world of sequential computation the most referenced algorithms for recognizing context–free languages are the CYK algorithm introduced in 1963 by Cocke, Younger and Kasami, and Earley's algorithm introduced in 1970. The former only deals with grammars in Chomsky normal form while the latter deals with every context–free grammar.

When in early 70's people started investigating parallel algorithms, immediately it appeared that obtaining efficient parallel parsers was not easy. In fact in 1977 Jones et al. [JL77] proved that the construction of a context–free recognizer is a P–complete problem.

In [Ruz80, Ryt86] however Ruzzo and Rytter showed that when we restrict our attention to context–free grammars in Chomsky normal form, the P–completeness result just mentioned can be overcome. In fact a parallel version of CYK can be obtained using Ruzzo and Rytter results.

On the other hand, Jones et al. result seems to deny the possibility of having a "parallel equivalent" of Earley's algorithm.

In this paper we show that this consideration is not true. More precisely we will show how to obtain a parallel algorithm for recognizing a language specified by a whatever context–free grammar. Using such an algorithm we will further describe a parallel algorithm for the parsing phase. To our knowledge these are the first parallel algorithms which solve such problems.

---

[*] This work was supported in part by the ESPRIT Basic Research Action No. 6317: "Algebraic and Syntactic Methods in Computer Science (ASMICS 2)" and by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST).

1 Introduction

In 1963 Cocke, Younger and Kasami independently discovered the first polynomial time algorithm (hereafter called CYK) for recognizing context–free languages (see [AU72] for a detailed description). But CYK has a limitation: it only deals with grammars in Chomsky normal form. We briefly recall that even if every context–free grammar can be efficiently transformed into an equivalent grammar in Chomsky normal form, such a transformation is quite unfair when the recognition process is followed by a parsing phase. In fact the parse tree which can be generated by a transformed grammar has very few resemblances to the parse tree related to the original grammar. Thus, for example, recovering *correct* semantic actions from the parse tree related to a transformed grammar is a very, if not impossible, hard task.

The problem of defining a recognizer and hence a parser which was able to deal with general context–free grammars was solved in 1970 by Earley [Ear70] with the introduction of a new recognizing algorithm obtained as an extension of Knuth's LR(k) method [Knu65]. For its generality Earley's algorithm is much more referenced than CYK especially for designing parsing algorithms for applications in which programming languages are not directly involved such as natural language processing [Tom85] and speech recognition [LS74].

At this point it is important to recall the attention of the reader on the following fact. Both CYK and Earley's algorithms are *uniform* with respect to the grammar, this means that the recognition process which they perform is not bounded to an *a priori* known grammar, but is designed for working with grammars which can be part of the input.

In early 70's with the diffusion of the first models of parallel computation people started investigating parallel algorithms. Quite soon, however, it appeared that the task of building parallel algorithms for recognizing context–free languages was not easy. In fact in 1977 Jones et al. [JL77] proved that the problem of the uniform recognition of context–free languages is P–complete [KR90], thus quite hardly parallelizable.

In 1980 however Ruzzo showed that when we restrict our attention to grammars in Chomsky normal form the P–completeness result could be overcome. In fact in [Ruz80] he described a parallel recognizer which can be executed on a CRCW PRAM in time $O(\log n)$ where n is the length of the input string. Subsequently, Rytter [Ryt86] referring to Ruzzo's result showed how to build a parallel parser for context–free grammars in Chomsky normal form.

Essentially Ruzzo and Rytter results lead us to a parallel version of CYK; on the other hand, Jones et al. result seems to deny the possibility of having a "parallel equivalent" of Earley's algorithm.

In this paper we show that this consideration is not true. In fact we show that if we renounce to the possibility of having a uniform recognizer, i.e., to the possibility of specifying the grammar in the input, then we can have a parallel algorithm for recognizing each kind of context–free grammar. More precisely, combining some revisited sequential versions of Earley's algorithm [GHR80, Kil85, Lei90] with parallel tree contraction techniques recently introduced in the theory of parallel algorithms [MR85, MRK88], we will build a parallel recognizer which works for every *fixed* but *arbitrary* context–free grammar. Using such an algortihm we will further show how to derive a parallel algorithm for parsing a generic context–free grammar.

The paper is organized as follows. In Section 2 we introduce notations and notions used throughout the paper. For ease of exposition initially we will consider context–free grammars with no unit and ϵ–productions. In particular in Section 3 we show a parallel recognizer for such grammars. Since a recognizer is generally simpler than a parser, this will permits us to better illustrate some general concepts required for the parallel parser design. In Section 4 we will prove the correctness of the algorithm presented in Section 3. In Section 5, we will show how to derive a parallel parser from the recognizer constructed in Section 3. Finally in Section 6 we describe how the parser so constructed can be extended to context–free grammars with both unit and ϵ–productions.

2 Preliminary definitions

We assume that the reader is familiar with notions and formalisms from formal language theory as reported in [HU79]. A context–free grammar G will be denoted by (V, Σ, P, S), where V is the set of variables or nonterminal, Σ is the set of terminal symbols, P is the finite set of productions, and S is the start symbol. We will suppose that the sets V and Σ are disjoint. Throughout the paper, variables from V will be denoted by the capital letters $A, B, C, \ldots$, terminal symbols by the lower case letters $a, b, c, \ldots$, strings from $(V \cup \Sigma)^*$ by the greek letters $\alpha, \beta, \gamma, \ldots$. ϵ will denote the empty string. Given a production $(A \rightarrow \alpha\beta) \in P$, a *dotted rule* is a string $A \rightarrow \alpha \cdot \beta$, where $\cdot \notin V \cup \Sigma$. The only dotted rule corresponding to the ϵ–production $A \rightarrow \epsilon$ will be denoted by $A \rightarrow \cdot$. $\alpha \overset{*}{\Rightarrow} \beta$ (α derives β) will denote the fact that β follows from α by zero or more applications of some productions of P. The number of these applications is called the *length* of the derivation.

The language generated by a grammar G will be denoted by $L(G)$ or L if G is understood. A *recognizer* is a procedure which accepts those strings in $L(G)$ and rejects all others. A *parser* is a recognizer which additionally outputs a parse tree or a derivation of each accepted input. It is convenient to fix $n \geq 1$, and to fix the input string to be parsed as $x = x_1 \ldots x_n$, $x \neq \epsilon$, $x_i \in \Sigma$, $i = 1, \ldots, n$.

Our model of parallel computation is a parallel random–access machine with both concurrent read and concurrent write, (CRCW PRAM)[KR90]. Such a machine consists of several independent sequential processors, each with its own private memory. Communication among processors is through shared memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM instruction, and write into one global or local memory location. In our particular case simultaneous write are allowed, and write conflicts are solved assuming that all processors writing into the same location must write the same value (the COMMON model).

3 The recognizer

In designing our algorithm, the main difficulty to overcome has been that of finding a strategy for recognizing in parallel general context–free grammars. This section is dedicated to illustrate the method we have devised for designing such an algorithm. In describing our algorithm we will refer to Earley's algorithm as presented in [Lei90]. Initially we will give an informal description of the algorithm; subsequently it will be formalized using a Concurrent Pascal like language.

Given a context–free grammar $G = (V, \Sigma, P, S)$, our algorithm performs the recognition of an input string x, $|x| = n$, by considering *items* of the form $(A \to \alpha \cdot \beta, i, j)$, where $A \to \alpha\beta \in P$, $0 \leq i \leq j \leq n$, and identifying among them *realizable items.* An item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ is said to be realizable if and only if $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$.[1] So, *the algorithm accepts the input string x if and only if there is a production $S \to \alpha$ such that the item $(S \to \alpha\cdot, 0, n)$ is realizable.*

Realizable items can be computed sequentially as follows. Initially all items of the form $(A \to \cdot\beta, i, i)$ are marked as realizable. Subsequently new items are marked as realizable using the following operations:

1. *scan:* if $x_j = a$ and $(A \to \alpha \cdot a\beta, i, j-1)$ is realizable then $(A \to \alpha a \cdot \beta, i, j)$ becomes realizable;
2. *complete:* if both items $(A \to \alpha \cdot B\beta, i, k)$ and $(B \to \gamma\cdot, k, j)$ are realizable, then the item $(A \to \alpha B \cdot \beta, i, j)$ becomes realizable.

Intuitively the *scan* operation augments the portion of a production recognized within an item when there is a matching between the first symbol of the portion of the production not yet recognized, and the next input symbol. The *complete* operation links together adjacent portions of the input string which have been recognized. We will now show how to parallelize such operations. The hardest part in such a task is the parallelization of the operation *complete*,[2] which we have realized referring to *partial evaluation* techniques [Ruz80, Ryt85, MR85, MRK88].

---

[1] Intuitively the dot in a realizable item indicates the portion of the right part of the production recognized so far.

[2] Note that a trivial parallelization of such an operation, obtained considering in parallel all triples of items $\mathcal{I} = (A \to \alpha \cdot B\beta, i, k)$, $\mathcal{J} = (B \to \gamma\cdot, k, j)$ and $\mathcal{K} = (A \to \alpha B \cdot \beta, i, j)$ and marking the item $\mathcal{K}$ as realizable if both items $\mathcal{I}$ and $\mathcal{J}$ are realizable, is not efficient. For example with such a parallel implementation, the *complete* operation would require linear time when working on the grammar with the following productions $S \to AS$, $S \to A$, $A \to a$.

More precisely, the algorithm we have devised, works in the following way. Given the grammar G, it starts considering a completely unconnected graph whose nodes are all possible items; during its execution the algorithm "*draws*" arcs between items and *marks* as realizable some of these items.

An arc $(\mathcal{I}, \mathcal{J})$ is drawn from item $\mathcal{I}$ to item $\mathcal{J}$ according to the following rules:

1. *parallel scan:* for all integers i, j, $0 \leq i \leq j \leq n$, do in parallel:
   if $x_j = a$ then connect $(A \to \alpha \cdot a\beta, i, j-1)$ to $(A \to \alpha a \cdot \beta, i, j)$;

2. *partial complete:* for each node in the graph do in parallel:
   if $(A \to \alpha \cdot B\beta, i, k)$ is realizable then connect $(B \to \gamma \cdot, k, j)$ to $(A \to \alpha B \cdot \beta, i, j)$;
   if $(B \to \gamma \cdot, k, j)$ is realizable then connect $(A \to \alpha \cdot B\beta, i, k)$ to $(A \to \alpha B \cdot \beta, i, j)$.

Note that if the arc $(\mathcal{I}, \mathcal{J})$ exists then the realizability of $\mathcal{I}$ implies the realizability of $\mathcal{J}$. Thus, if after the arc $(\mathcal{I}, \mathcal{J})$ has been drawn, the algorithm will mark the item $\mathcal{I}$ as realizable, then also the item $\mathcal{J}$ will be marked as realizable (this step will be performed by the operation *mark* defined later). Note also that when both the arcs $(\mathcal{I}, \mathcal{J})$ and $(\mathcal{J}, \mathcal{K})$ have been "drawn", then also the arc $(\mathcal{I}, \mathcal{K})$ can be added. Thus, we introduce besides the operations just defined the additional operation *square* which computes transitive arcs of the graph.

Nodes are marked as realizable according to the following rules:

1. *parallel predict:* for each integer i, $0 \leq i \leq n$, do in parallel:
   mark as realizable all items of the form $(A \to \cdot \beta, i, i)$;

2. *mark:* for each item $\mathcal{I}$ do in parallel:
   if $\mathcal{I}$ is realizable then mark as realizable all items connected to $\mathcal{I}$.

It is not difficult to see that the operations *parallel predict* and *parallel scan* can be executed only once, at the start of the recognition process, while we will show that the operations *partial complete*, *square* and *mark* have to be executed a logarithmic (in the length of the input) number of times in order to get a correct result.

The reader acquainted with parallel tree contraction techniques should have recognized the strong analogy between the procedures *Rake* and *Compress* introduced in [MR85] or the procedures *MM*, $Eval_+$ and $Eval_\times$ introduced in [MRK88] and our operations *partial complete, square* and *mark*. In fact, *square* is a compression operation, while the operations *partial complete* and *mark* "remove leaves".

A brief example will now illustrate how the algorithm works. Suppose we have to recognize the input string $x = a + a * a$, with respect to the grammar G, whose variables are E, T, P, whose terminal symbols are $a, +, *$, whose start symbol is E, and whose productions are:

$$\begin{array}{l} E \to T \\ E \to E + T \\ T \to P \\ T \to T * P \\ P \to a \end{array}$$

In the first step of the algorithm the following items are marked as realizable, by the *parallel predict* operation, for $i = 0, \ldots, 5$:

$$\begin{array}{lll} (E \to \cdot T, & i, & i) \\ (E \to \cdot E + T, & i, & i) \\ (T \to \cdot P, & i, & i) \\ (T \to \cdot T * P, & i, & i) \\ (P \to \cdot a, & i, & i). \end{array}$$

Subsequently the operation *parallel scan* draws the following arcs:

$$\begin{array}{llllllll} ((P \to \cdot a, & 0, & 0), & (P \to a \cdot, & 0, & 1)) & & \\ ((E \to E \cdot +T, & i, & 1), & (E \to E + \cdot T, & i, & 2)), & i = 0, 1 \\ ((P \to \cdot a, & i, & 2), & (P \to a \cdot, & i, & 3)), & i = 0, 1, 2 \\ ((T \to T \cdot *P, & i, & 3), & (T \to T * \cdot P, & i, & 4)), & i = 0, 1, 2, 3 \\ ((P \to \cdot a, & i, & 4), & (P \to a \cdot, & i, & 5)), & i = 0, 1, 2, 3, 4 \end{array}$$

At this point the algorithm starts iterating the sequence of operations *partial complete*, *square* and *mark*. In the first of such iterations the following arcs are added by the *partial complete* operation, for $0 \leq i \leq j \leq 5$:

$$\begin{array}{llllll} ((T \to P \cdot, & i, & j), & (E \to T \cdot, & i, & j)) \\ ((T \to T * P \cdot, & i, & j), & (E \to T \cdot, & i, & j)) \\ ((E \to T \cdot, & i, & j), & (E \to E \cdot +T, & i, & j)) \\ ((E \to E + T \cdot, & i, & j), & (E \to E \cdot +T, & i, & j)) \\ ((P \to a \cdot, & i, & j), & (T \to P \cdot, & i, & j)) \\ ((T \to P \cdot, & i, & j), & (T \to T \cdot *P, & i, & j)) \\ ((T \to T * P \cdot, & i, & j), & (T \to T \cdot *P, & i, & j)); \end{array}$$

while the following items are marked as realizable by the operation *mark*:

$$\begin{array}{lll} (P \to a \cdot, & 0, & 1) \\ (P \to a \cdot, & 2, & 3) \\ (P \to a \cdot, & 4, & 5) \end{array}$$

In the second iteration of the algorithm, the items

$$\begin{array}{lll} (E \to E \cdot +T, & 0, & 1) \\ (E \to E + \cdot T, & 0, & 2) \\ (T \to T \cdot *P, & 2, & 3) \\ (T \to T * \cdot P, & 2, & 4) \\ (T \to T * P \cdot, & 2, & 5) \end{array}$$

are marked as realizable by the operation *mark*. Moreover, in the third iteration also the item $(E \to E + T\cdot, 0, 5)$ is marked as realizable, thus proving the membership of x to $L(G)$.

In summary, the algorithm we have just outlined can be described in the following way.

Algorithm 1

```
begin
(1) parallel predict
(2) parallel scan
repeat $O(\log n)$ times
    begin
(3)     partial complete
(4)     square
(5)     mark
    end
(6) answer
end
```

The operations contained in the algorithm can formally be defined in the following way. Consider two boolean arrays R and Z. At the start of the computation all components of R and Z are initialized to FALSE. The first array is indexed by items and it is used to mark them as realizable, i.e., at every step of the computation $R(\mathcal{I}) =$ TRUE if and only if the item $\mathcal{I}$ has been marked as realizable. The array Z is a two–dimensional array indexed by items and used as adjacency matrix of the graph which the algorithm constructs during its execution. In particular $Z(\mathcal{I}, \mathcal{J}) =$ TRUE if and only if an arc has been drawn connecting item $\mathcal{I}$ to item $\mathcal{J}$. The algorithm accepts the input string x if and only if at the end of the computation a boolean variable *output* contains the value TRUE.

- *parallel predict*

```
for all $(A \to \cdot\beta, i, i)$ in parallel do $R(A \to \cdot\beta, i, i) :=$ TRUE
```

- *parallel scan*

```
for all $(A \to \alpha a \cdot \beta, i, j)$ in parallel do
        if $x_j = a$ then $Z((A \to \alpha \cdot a\beta, i, j-1), (A \to \alpha a \cdot \beta, i, j)) :=$ TRUE
```

- *partial complete*

```
for all $(A \to \alpha \cdot B\beta, i, k)$ and for all $(B \to \gamma\cdot, k, j)$ in parallel do
begin
        if $R(A \to \alpha \cdot B\beta, i, k)$ then $Z((B \to \gamma\cdot, k, j), (A \to \alpha B \cdot \beta, i, j)) :=$ TRUE
        if $R(B \to \gamma\cdot, k, j)$ then $Z((A \to \alpha \cdot B\beta, i, k), (A \to \alpha B \cdot \beta, i, j)) :=$ TRUE
end
```

- *square*

  ```
  for all $\mathcal{I}, \mathcal{J}, \mathcal{K}$ in parallel do if $Z(\mathcal{I}, \mathcal{J})$ and $Z(\mathcal{J}, \mathcal{K})$ then $Z(\mathcal{I}, \mathcal{K}) :=$ TRUE
  ```

- *mark*

  ```
  for all $\mathcal{I}, \mathcal{J}$ in parallel do if $Z(\mathcal{I}, \mathcal{J})$ and $R(\mathcal{I})$ then $R(\mathcal{J}) :=$ TRUE
  ```

- *answer*

  ```
  output := FALSE
  for all $\mathcal{I} = (S \to \alpha\cdot, 0, n)$ in parallel do
      if $R(\mathcal{I})$ then output := TRUE
  ```

4 Correctness proof and complexity analysis

In this section we will show that given a context–free grammar G, Algorithm 1 recognizes exactly all strings $x \in L(G)$, in time $O(\log n)$ using at most $O(n^6)$ processors.

The proof that every string accepted by the algorithm belongs to $L(G)$ is quite immediate, the proof of the opposite implication is instead much more involved. We realized such a proof referring to the notion of path system introduced by Cook in [Coo70] and using the *balanced divide–and–conquer* technique introduced by Gurari and Ibarra in [GI80].

Lemma 4.1 *After every step of Algorithm 1 the following assertions hold:*

(i) *for all items* $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ *and* $\mathcal{J} = (C \to \gamma \cdot \delta, p, q)$, *if* $\mathcal{J}$ *is connected to* $\mathcal{I}$, *i.e.,* $Z(\mathcal{J}, \mathcal{I}) =$ TRUE, *then* $i \leq p \leq q \leq j$ *and* $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_p \gamma x_{q+1} \dots x_j$*;*

(ii) *for every item* $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, *if* $(A \to \alpha \cdot \beta, i, j)$ *is realizable, i.e.,* $R(\mathcal{I}) =$ TRUE*, then* $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$*.*

Proof. Statement (i) trivially holds if the value TRUE is assigned to $Z(\mathcal{J}, \mathcal{I})$ in step (2) of Algorithm 1. It is also clear that (ii) holds if the value TRUE is assigned to $R(\mathcal{I})$ in step (1).

Consider now two items $\mathcal{I}, \mathcal{J}$ and assume that for such items (i) and (ii) hold before the execution of steps (3), (4) and (5). The following cases have to be considered.

- $Z(\mathcal{J}, \mathcal{I})$ becomes TRUE executing step (3); this can happen only when
    1. $\mathcal{I} = (A \to \alpha B \cdot \beta, i, j)$, $\mathcal{J} = (B \to \gamma\cdot, k, j)$ and $R(\mathcal{K}) =$ TRUE, where $\mathcal{K} = (A \to \alpha \cdot B\beta, i, k)$. By (ii) this implies $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_k$ and then $\alpha B \overset{*}{\Rightarrow} x_{i+1} \dots x_k B \Rightarrow x_{i+1} \dots x_k \gamma$.

2. $\mathcal{I} = (A \to \alpha B \cdot \beta, i, j)$, $\mathcal{J} = (A \to \alpha \cdot B\beta, i, k)$ and $R(\mathcal{K}) = \text{TRUE}$, where $\mathcal{K} = (B \to \gamma \cdot, k, j)$. By (ii) this implies $\gamma \overset{*}{\Rightarrow} x_{k+1} \dots x_j$ and then $\alpha B \Rightarrow \alpha\gamma \overset{*}{\Rightarrow} \alpha x_{k+1} \dots x_j$.

- $Z(\mathcal{J}, \mathcal{I})$ becomes TRUE executing step (4). This case can be easily verified.

- $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ is marked as realizable executing step (5). This implies that there exists an item $\mathcal{J} = (C \to \gamma \cdot \delta, p, q)$ for which $Z(\mathcal{J}, \mathcal{I})$ and $R(\mathcal{J})$ hold. Thus, $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_p \gamma x_{q+1} \dots x_j$ and $\gamma \overset{*}{\Rightarrow} x_{p+1} \dots x_q$. Then, we can conclude that $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$. ∎

As a consequence of the statement (ii) of previous lemma, it is immediate to conclude that every string x accepted by Algorithm 1 belongs to the language $L(G)$. The converse result is more complicated. To prove it we need the preliminary notion of recognition tree.

Definition 4.1 *Given a context-free grammar $G = (V, \Sigma, P, S)$ and an input string $x = x_1 \dots x_n$, $x_i \in \Sigma$, a* partial recognition tree *of x is a binary tree, whose nodes are labeled either with items or with pairs $(a, j) \in \Sigma \times \{1, \dots, n\}$ such that:*

1. *the label of an* internal node *is an item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, with $\alpha \neq \epsilon$; the label of its left son is $\mathcal{J}$ and the label of its right son is $\mathcal{K}$, where:*
    - *if $\alpha = \alpha' B$, with $B \in V$, then $\mathcal{J} = (A \to \alpha' \cdot B\beta, i, k)$ and $\mathcal{K} = (B \to \gamma \cdot, k, j)$, for some k, $i \leq k \leq j$, and $B \to \gamma \in P$;*
    - *if $\alpha = \alpha' a$, with $a \in \Sigma$, then $j > i$, $a = x_j$, $\mathcal{J} = (A \to \alpha' \cdot a\beta, i, j-1)$ and $\mathcal{K} = (a, j)$;*
2. *the labels of the leaves are items of the form $(A \to \cdot\alpha, i, i)$ and, from left to right, the pairs $(x_{p+1}, p+1), \dots, (x_q, q)$, $0 \leq p \leq q \leq n$.*

A recognition tree *of x is a partial recognition tree whose root is labeled with an item $(S \to \alpha \cdot, 0, n)$, for some production $S \to \alpha$. The number of nodes of a partial recognition tree T will be denoted by $\eta(T)$.*

We underline that while parse trees describe the syntactical structure of sentences of a language $L(G)$ with respect to the grammar G, recognition trees are related, as we will show in the following, to the recognition process performed by Algorithm 1.

A recognition tree of the string $x = a + a * a$, with respect to the grammar G considered in the previous section, is represented in Figure 1.

Note that for grammars G which are not *cycle-free*, i.e., grammars which contain a derivation of the form $A \overset{+}{\Rightarrow} A$ for some variable A, the number of recognition trees related to some string $x \in L(G)$ can be unbounded. However next theorem shows that even if a string x belonging to $L(G)$ has infinitely many recognition trees, it has at least one recognition tree whose number of nodes is linear in the length of x.

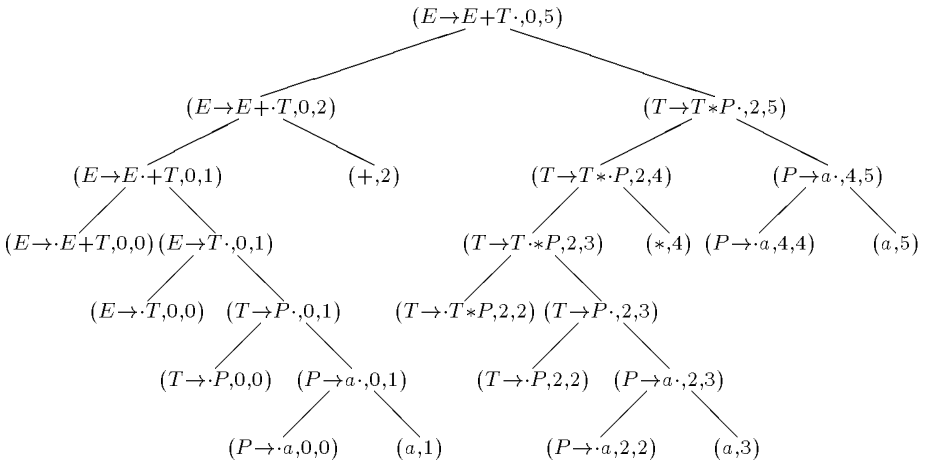

Figure 1: a recognition tree of the string $x = a + a * a$

Theorem 4.1 *Given a context-free grammar* $G = (V, \Sigma, P, S)$*, for every string* $x = x_1 \dots x_n$*,* $x_k \in \Sigma$*,* $k = 1, \dots, n$*, and for every item* $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$*, if* $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$ *then there exists a partial recognition tree* T *of* x*, with root labeled* $\mathcal{I}$ *and whose number of nodes is* $O(j - i)$*.*

Proof. Given an item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, let $\tau(\mathcal{I})$ be the length of a shortest derivation $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$. Then, we define the *rank* of $\mathcal{I}$ as $rank(\mathcal{I}) = \tau(\mathcal{I}) + j - i + 1$.

First, we prove that for every item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, if $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$ then there exists a partial recognition tree T of x, whose root is labeled $\mathcal{I}$ and such that $\eta(T) < 2rank(\mathcal{I})$. The proof is by induction on $rank(\mathcal{I})$.

Basis. If $rank(\mathcal{I})$ is 1 then $j = i$ and $\alpha = \epsilon$; the partial recognition tree T contains only one node labeled $(A \to \cdot\beta, i, i)$, and $\eta(T) = 1$.

Induction Step. Let $\mathcal{I}$ be an item with $rank(\mathcal{I}) > 1$. First, observe that in this case if $\alpha = \epsilon$ then $j > i$ so α cannot generate the string $x_{i+1} \dots x_j$. Thus we will only consider $\alpha \neq \epsilon$. Two cases arise, depending on whether α ends in a terminal or ends in a variable.

Case 1: $\alpha = \alpha' a$ for some $a \in \Sigma$. Since $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$, we have $a = x_j$, and $\alpha' \overset{*}{\Rightarrow} x_{i+1} \dots x_{j-1}$. Consider the item $\mathcal{I}' = (A \to \alpha' \cdot a\beta, i, j-1)$. We have $rank(\mathcal{I}') = rank(\mathcal{I}) - 1$. By induction hypothesis, there exists a partial recognition tree T' with root labeled $\mathcal{I}'$ such that $\eta(T') < 2rank(\mathcal{I}')$. Consider the tree T with root labeled $\mathcal{I}$, left subtree T' and right subtree the only node labeled (a, j). It is not difficult to see that T is a partial recognition tree satisfying our requirements.

Case 2: $\alpha = \alpha' B$ for some B in V. There is some k, $i \leq k \leq j$, such that $\alpha' \overset{*}{\Rightarrow} x_{i+1} \dots x_k$ and $B \overset{*}{\Rightarrow} x_{k+1} \dots x_j$. Consider the item $\mathcal{I}' = (A \to \alpha' \cdot B\beta, i, k)$. Then, $\tau(\mathcal{I}') < \tau(\mathcal{I})$ and $rank(\mathcal{I}') < rank(\mathcal{I})$. By induction hypothesis, we can find a recognition tree T' with root labeled $\mathcal{I}'$ such that $\eta(T') < 2rank(\mathcal{I}')$. Let $B \Rightarrow \gamma$

be the first step in one of the shortest derivation of $B \overset{*}{\Rightarrow} x_{k+1} \dots x_j$, and let $\mathcal{I}''$ be the item $(B \to \gamma \cdot, k, j)$. Then $\tau(\mathcal{I}) = \tau(\mathcal{I}') + \tau(\mathcal{I}'') + 1$ and $rank(\mathcal{I}'') < rank(\mathcal{I})$. By induction hypothesis, there exists a partial recognition tree T'' with root $\mathcal{I}''$ and $\eta(T'') < 2rank(\mathcal{I}'')$. The tree T with root $\mathcal{I}$, left subtree T' and right subtree T'' is a partial recognition tree of $x_{i+1} \dots x_j$, moreover $\eta(T) = \eta(T') + \eta(T'') + 1$. Then, since $\eta(T'') + 1 \leq 2rank(\mathcal{I}'')$, we obtain $\eta(T) < 2rank(\mathcal{I}') + 2rank(\mathcal{I}'') = 2rank(\mathcal{I})$. This completes the induction. For concluding the proof we recall that for every context–free grammar G there exists a constant c such that if a string $y \neq \epsilon$ can be derived from a variable A, then there exists a derivation $A \overset{*}{\Rightarrow} y$ of length at most $c|y|$. The value of the constant c can be found in [Sip82].

Let $\mathcal{I}$ be the item $(A \to \alpha \cdot \beta, i, j)$ with $i \neq j$, and T the partial recognition tree with root labeled $\mathcal{I}$ obtained using the previous induction. Then, we have $\tau(\mathcal{I}) \leq c(j-i)$. This permit us to conclude that $\eta(T) = O(j-i)$. ∎

We point out that if a recognition tree T of a string x contains two nodes v and z marked with the same item $\mathcal{I}$, then z must be a descendant of v (or vice versa). Thus, the tree T' obtained from T substituting the subtree rooted at v with the subtree rooted at z is also a recognition tree of x. In this way, given a string $x \in L(G)$, it is always possible to find a recognition tree whose nodes are all differently labeled. It is possible to verify that this property holds for the recognition tree constructed in the proof of Theorem 4.1. Then, from this point on, we only consider recognition trees such that all the labels of nodes are different. This will permit us of identifying every node with its label and of writing *the node* $\mathcal{I}$ instead than *the node whose label is* $\mathcal{I}$.

The next theorem is crucial in order to prove the correctness of Algorithm 1. Its proof is obtained referring to the following lemma and using the *balanced divide–and–conquer* technique introduced in [GI80] for evaluating path systems.

Lemma 4.2 [Ruz80] *Let T be a tree having $z > 3$ nodes, with each node having at most two descendants. Then there is a node s in T such that $z/12 < z_1, z_2 \leq 11z/12$, where z_1 is the number of descendants of s (including s), and z_2 is the number of nodes which are not proper descendants of s (again including s, so $z_1 + z_2 = z + 1$).*[3]

In the following, given a partial recognition tree T, a node $\mathcal{I}$ of T and a descendant $\mathcal{J}$ of $\mathcal{I}$, we will denote by $T(\mathcal{I})$ the subtree of T whose root is $\mathcal{I}$ and which contains all descendants of $\mathcal{I}$ in T, and by $T(\mathcal{I}, \mathcal{J})$ the tree obtained removing from the tree $T(\mathcal{I})$ all descendants of $\mathcal{J}$ (except $\mathcal{J}$).

Theorem 4.2 *Let x be a string belonging to $L(G)$, where G is some fixed context–free grammar, and let T be a recognition tree of x. Given an item $\mathcal{I}$, if $\mathcal{I}$ is a node of T then the following statements hold:*

(1) *Algorithm 1 set to* TRUE *$R(\mathcal{I})$ within $3\log_{12/11} \eta(T(\mathcal{I}))$ iterations;*

---

[3] Observe that the node s cannot be the root or a leaf of T.

(2) *for every item $\mathcal{J}$, $\mathcal{I} \neq \mathcal{J}$, if $\mathcal{J}$ is a descendant of $\mathcal{I}$ in T, then Algorithm 1 sets to* TRUE $Z(\mathcal{J},\mathcal{I})$ *within* $3\log_{12/11}\eta(T(\mathcal{I},\mathcal{J}))$ *iterations.*

Proof. By induction on the number z of nodes of the trees.

Basis ($z \leq 3$). First, we prove (1). If $\eta(T(\mathcal{I})) = 1$ then $\mathcal{I}$ is a leaf of the form $(A \to \cdot\beta, i, i)$ and TRUE is assigned to $R(\mathcal{I})$ by the operation *parallel predict*, without any iteration. We observe that there are no recognition trees with an even number of nodes. Then, the next case we have to consider is $\eta(T(\mathcal{I})) = 3$. In this case $\mathcal{I}$ has the form $(A \to a \cdot \beta, i-1, i)$ and its sons are the nodes $(A \to \cdot a\beta, i-1, i-1)$ and (a, i). We observe that TRUE is assigned to $R(A \to \cdot a\beta, i-1, i-1)$ and to $Z((A \to \cdot a\beta, i-1, i-1), (A \to a \cdot \beta, i-1, i))$ by the operations *parallel predict* and *parallel scan*, respectively. Then, in the first iteration, the instruction *mark* assign TRUE to $R(A \to a \cdot \beta, i-1, i)$.

For proving (2) the only case we have to consider is $\eta(T(\mathcal{I},\mathcal{J})) = 3$. In this case the root $\mathcal{I}$ of $T(\mathcal{I},\mathcal{J})$ has two sons $\mathcal{J}$ and $\mathcal{K}$ where $\mathcal{K}$ is a leaf of T. If $\mathcal{K}$ is the right son of $\mathcal{I}$ then $\mathcal{K}$ is of the form (a, i) and then $Z(\mathcal{J},\mathcal{I})$ is set to TRUE by *parallel scan*. Otherwise, $\mathcal{K}$ has the form $(A \to \cdot\beta, i, i)$, TRUEis assigned to $R(\mathcal{K})$ by *parallel predict* and then, in the first iteration, the operation *partial complete* sets $Z(\mathcal{J},\mathcal{I})$ to TRUE.

Induction Step. Let be $z = \eta(T(\mathcal{I}))$. By Lemma 4.2, there exists a node $\mathcal{K}$ of $T(\mathcal{I})$ such that both trees $T(\mathcal{I},\mathcal{K})$ and $T(\mathcal{K})$ have at most $11z/12$ nodes. Of course, $\mathcal{I} \neq \mathcal{K}$. Then, by induction hypothesis, it turns out that within $3\log_{12/11}(11z/12) = 3\log_{12/11} z - 3$ iterations of Algorithm 1, both $Z(\mathcal{K},\mathcal{I})$ and $R(\mathcal{K})$ become TRUE. In the next iteration, the operation *mark* will set to TRUE the value of $R(\mathcal{I})$. This proves (1).

Suppose now $z = \eta(T(\mathcal{I},\mathcal{J}))$. By Lemma 4.2, it turns out that there exists a node $\mathcal{K}$ of $T(\mathcal{I},\mathcal{J})$, $\mathcal{K} \neq \mathcal{I}$, $\mathcal{K} \neq \mathcal{J}$, such that both the subtree of $T(\mathcal{I},\mathcal{J})$ rooted at $\mathcal{K}$ and the tree obtained by deleting from $T(\mathcal{I},\mathcal{J})$ all proper descendants of $\mathcal{K}$ have at most $11z/12$ nodes. Now, two cases arise depending on whether $\mathcal{J}$ is or is not a descendant of $\mathcal{K}$.

$\mathcal{J}$ is a descendant of $\mathcal{K}$. In this case the tree $T(\mathcal{I},\mathcal{J})$ can be split in the two trees $T(\mathcal{I},\mathcal{K})$ and $T(\mathcal{K},\mathcal{J})$ each containing at most $11z/12$ nodes. As in the proof of (1), using the induction hypothesis, we can conclude that within $3\log_{12/11} z - 3$ iterations both variables $Z(\mathcal{K},\mathcal{I})$ and $Z(\mathcal{J},\mathcal{K})$ become TRUE. Thus, in the subsequent iteration, the operation *square* will set to TRUE the value of $Z(\mathcal{J},\mathcal{I})$.

$\mathcal{J}$ is not a descendant of $\mathcal{K}$. Let $\mathcal{I}'$ be the first (from the leaves to the root) common predecessor of $\mathcal{J}$ and $\mathcal{K}$, and denote by $\mathcal{J}'$ the son of $\mathcal{I}'$ on the path from $\mathcal{I}'$ to $\mathcal{J}$ and by $\mathcal{K}'$ the son of $\mathcal{I}'$ on the path from $\mathcal{K}'$ to $\mathcal{K}$ (see Figure 2). Consider now the trees $T(\mathcal{K})$, $T(\mathcal{K}',\mathcal{K})$, $T(\mathcal{J}',\mathcal{J})$ and $T(\mathcal{I},\mathcal{I}')$. Again, by Lemma 4.2 all these trees have at most $11z/12$ nodes. Then, by induction hypothesis, after $3\log_{12/11} z - 3$ iterations of Algorithm 1, all variables $R(\mathcal{K})$, $Z(\mathcal{K},\mathcal{K}')$, $Z(\mathcal{J},\mathcal{J}')$ and $Z(\mathcal{I}',\mathcal{I})$ will contain the value TRUE.

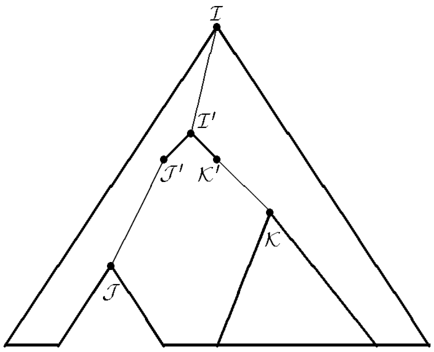

Figure 2: decomposition of the tree $T(\mathcal{I}, \mathcal{J})$.

In the iteration the operation *mark* will set $R(\mathcal{K}')$ to TRUE, in the following one the operation *partial complete* sets to TRUE $Z(\mathcal{J}', \mathcal{I}')$, and the operation *square* sets to TRUE $Z(\mathcal{J}', \mathcal{I})$. Finally, with another iteration $Z(\mathcal{J}, \mathcal{I})$ is set to TRUE by the operation *square*. Thus, within $3 \log_{12/11} z$ iterations, $Z(\mathcal{J}, \mathcal{I})$ becomes TRUE.[4] ∎

Theorem 4.3 *Given a context-free grammar G, Algorithm 1 decides the membership to $L(G)$ of an input string x, of length n, in time $O(\log n)$ using $O(n^6)$ processors on a CRCW PRAM.*

Proof. First, we observe that as an immediate consequence of Lemma 4.1, every string accepted by Algorithm 1 belongs to $L(G)$. Now, we prove the converse. Let be $x \in L(G)$, then, by Theorem 4.1 there exists a recognition tree T with $O(|x|)$ nodes. Applying Theorem 4.2, it turns out that after $O(\log |x|)$ iterations, Algorithm 1 is able to assign the value TRUE to the item labeling the root of T. Then x is accepted.

The most expensive steps contained in Algorithm 1 correspond to the operations *partial complete* and *square*, which in parallel work on all triples of items. Since the number of items is $O(|x|^2)$, the number of processors required by such operations is $O(|x|^6)$. ∎

---

[4] This part of the proof suggests some improvements to Algorithm 1. In fact, if in every iteration the sequence of operations *partial complete; square; mark* is substituted by the sequence *mark; partial complete; square; square*, then the constant 3 of the Theorem can be replaced by 1. Moreover, the basis of the logarithm can be increased using more accurate decompositions of trees than the one given by Lemma 4.2 (see for example [GI80]). However, since in this paper we are mainly interested in the asymptotical complexity of the algorithm, we do not deepen into these considerations.

5 The parser

Given the recognizer shown in Section 3, the construction of a general parser is relatively easy. In particular, for such a task we propose a generalization of a method introduced in [Ryt86] for grammars in Chomsky Normal Form.

The parser can be divided in two phases. In the first phase, given the input string x, and the array R "constructed" by the recognizer, we build a recognition tree for x; subsequently from such a recognition tree, a parse tree is recovered.

5.1 Construction of the recognition tree

In order to obtain a recognition tree for x from the array R computed by the recognizer, first we build a directed acyclic graph $\mathcal{G} = (V, E)$. The vertex set V of this graph is the union of the set of all realizable items with the set $\{(x_i, i) \mid i = 1, \ldots, n\}$. The set E of arcs is computed using the following algorithm:

Algorithm 2

```
begin
    for all items $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ in parallel do
        if $\alpha \neq \epsilon$ and $R(\mathcal{I})$ = TRUE then
        begin
            select a "possible pair" $(\mathcal{J}, \mathcal{K})$ of sons[5] of $\mathcal{I}$ s.t. $R(\mathcal{J})$ = TRUE
              and if $\mathcal{K}$ is an item then $R(\mathcal{K})$ = TRUE
            $E(\mathcal{J}, \mathcal{I}) := 1$
            $E(\mathcal{K}, \mathcal{I}) := 1$
        end
end
```

Given the grammar $G = (\Sigma = \{a, b\}, V = \{S, B\}, P, S)$, where the set P contains the productions $S \to Sa$, $S \to aB$, $B \to Ba$, $B \to b$, a graph $\mathcal{G}$ computable using Algorithm 2 is represented in Figure 3.

The following general statements hold:

Lemma 5.1 *The graph $\mathcal{G}$ above defined, related to a grammar G which does not contain ϵ–productions and unit productions, is acyclic.*

Proof. First, we remark the following facts.

---

[5] $(\mathcal{J}, \mathcal{K})$ is a "possible pair" of sons of $\mathcal{I}$ when $(\mathcal{J}, \mathcal{K})$ can be a pair of sons of $\mathcal{I}$ in some recognition tree according to Definition 4.1, i.e., either $\mathcal{I} = (A \to \alpha' B \cdot \beta, i, j)$, $\mathcal{J} = (A \to \alpha' \cdot B\beta, i, k)$, $\mathcal{K} = (B \to \gamma \cdot, k, j)$, for some $i \leq k \leq j$, $B \to \gamma \in P$, or $\mathcal{I} = (A \to \alpha' x_j \cdot \beta, i, j)$, $\mathcal{J} = (A \to \alpha' \cdot x_j \beta, i, j-1)$, $\mathcal{K} = (x_j, j)$.

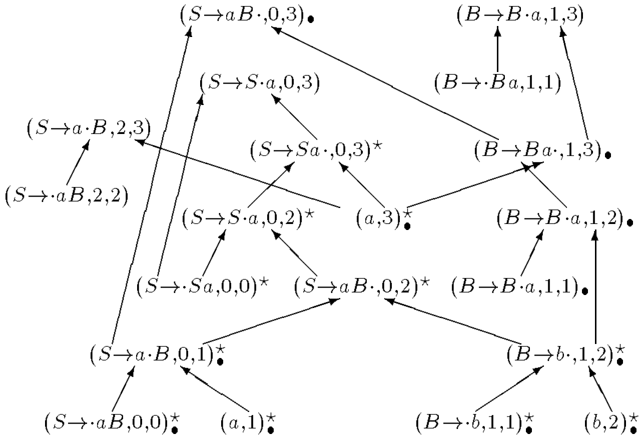

Figure 3: the graph $\mathcal{G}$. * indicates the nodes belonging to the recognition tree with root $(S \to Sa\cdot, 0, 3)$, while $\bullet$ indicates the nodes belonging to the recognition tree with root $(S \to aB\cdot, 0, 3)$. The nodes corresponding to items of the form $(A \to \cdot\beta, i, i)$ without connections are not represented.

(1) Every item of the form $(B \to \gamma\cdot, j, j)$ cannot be a node of a recognition tree.
In fact, every item labeling a node of a recognition tree is realizable; so if $(B \to \gamma\cdot, j, j)$ is a node of a recognition tree then $B \overset{*}{\Rightarrow} \epsilon$. This is not possible since G does not contain ϵ–productions.

(2) If the graph $\mathcal{G}$ contains a path from an item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ to an item $\mathcal{I}' = (A' \to \alpha' \cdot \beta', i', j')$, then $i \geq i'$ and $j \leq j'$.
This fact can be easily proved as a consequence of the definition of recognition tree.

Consider now a generic item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$. If $\alpha = \epsilon$ then there are no arcs entering in $\mathcal{I}$. So, $\mathcal{I}$ cannot belong to a loop. Suppose now that $\alpha \neq \epsilon$. We have to consider two cases, depending on whether α ends with a terminal symbol or with a variable.

(a) Suppose $\alpha = \alpha' x_j$, $\alpha' \in (V \cup \Sigma)^*$. In this case Algorithm 2 chooses $\mathcal{J} = (A \to \alpha' \cdot x_j\beta, i, j-1)$ and $\mathcal{K} = (x_j, j)$ as sons of $\mathcal{I}$. Using (2) it is immediate to see that in this case there are no paths from $\mathcal{I}$ to $\mathcal{J}$ in $\mathcal{G}$, thus $\mathcal{I}$ does not belong to a loop.

(b) Suppose now $\alpha = \alpha' B$, for some $\alpha' \in (V \cup \Sigma)^*$, $B \in V$, when the nodes selected by Algorithm 2 as sons of $\mathcal{I}$ have the following form: $\mathcal{J} = (A \to \alpha' \cdot B\beta, i, k)$, $\mathcal{K} = (B \to \gamma\cdot, k, j)$, $i \leq k \leq j$, $B \to \gamma \in P$.

From (1), it turns out that $k < j$