Bidirectional Parsing for Context Free Languages

Ştefan ANDREI [*], Manfred KUDLEK [†]

Contents

1 The left and right bidirectional parsers **2**

2 A parallel approach for (general) bidirectional parsing **16**

3 Deterministic subclasses of context free grammars **23**
3.1 $RR(k)$ grammars 24
3.2 $RL(k)$ grammars 31
3.3 SIP grammars 45

4 Deterministic bidirectional parsing for context free languages **50**

5 Conclusions **55**

Abstract

In this paper, we describe some subclasses of context-free grammars for which a parallel approach useful for solving the membership problem will be defined. More precisely, we will combine the classical type of parser attached to a grammar G with a "mirror" process for G. They analyse the input word from both sides using two processors.

In the first section, we present the (general) left and right bidirectional parsers which use a nondeterministic device for any context free language.

In the second section, a general SIMD model for describing bidirectional parsing is introduced. It contains two algorithms which use a backtracking method to describe the nondeterministic behaviour of the (general) left and right bidirectional model.

The third section treats some deterministic subclasses of context free grammars, such as $RR(k)$, $RL(k)$ and SIP grammars. The idea is to put in a "reverse view" the classical deterministic subclasses of context free grammars.

The fourth section points out the application of the (general) left and right bidirectional parser to the deterministic subclasses described in the

[*] Faculty of Informatics, "Al.I.Cuza" University, Str. Berthelot, nr. 16, 6600, Iaşi, România. E-mail: stefan@infoiasi.ro. This work was supported by The World Bank/Joint Japan Graduate Scholarship Program.

[†] Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany. E-mail: kudlek@informatik.uni-hamburg.de

previous section. Ten new classes of grammars obtained by combining the previous ones are defined, using descendant and ascendant strategies. The membership problem for all these type of grammars can be solved with a parallel algorithm in linear time.

Keywords: context-free grammars, bidirectional parsing, parallel algorithms

Mathematics Subject Classification: 68Q22, 68Q50, 68Q52, 68Q68.

1 The left and right bidirectional parsers

In this paper we define two parsers which have the main goal to accept the context free languages. The idea is similar - but more general - to that of [AnK98], The input word is also analysed from the both sides, but the algorithm works (having two "heads" which operate independently) not only for the class of linear grammars. That is why a parallel approach for these subclasses of grammars can be defined.

Before giving the definition of a new kind of parsers for context free grammars, we give a list of definitions and notations used in the paper.

Definitions:

- **context free grammar:** $G = (V_N, V_T, S, P)$, where:
  - V_N - the set of nonterminal symbols;
  - V_T - the set of terminal symbols;
  - $V = V_N \cup V_T$ - the set of symbols of G;
  - S - the start symbol;
  - $P \subseteq V_N \times V^*$ - the set of productions. A pair $(A, \beta) \in P$ is called an $A-$production and it is denoted by $A \to \beta$. The productions $A \to \beta_1,\ A \to \beta_2, ...,\ A \to \beta_k$ will be denoted by $A \to \beta_1 \,|\, \beta_2 \,|\, ... \,|\, \beta_k$ (sometimes).
- **empty word:** λ (the word of length 0);
- **derivation in G:** $\alpha \underset{G}{\Longrightarrow} \beta$ if $\exists\ A \in \alpha$ and $A \to r \in P$ such that

  $\alpha = \alpha_1\, A\, \alpha_2,\ \beta = \beta_1\, r\, \beta_2$; the transitive (reflexive) closure of the relation $\underset{G}{\Longrightarrow}$ is denoted by $\underset{G}{\overset{+}{\Longrightarrow}}$ ($\underset{G}{\overset{*}{\Longrightarrow}}$);
- a derivation is called **left most** (denoted by $\underset{lm}{\Longrightarrow}$) if in every sentential form (using a production from G) the first occurrence of a nonterminal symbol is replaced;

- a context-free grammar G is called **ambiguous** if there exists a word $w \in V_T^*$ for which there exist at least two distinct left most derivations $S \overset{*}{\underset{G}{\Longrightarrow}} w$;
- X is an **accessible symbol in** G if there exists a derivation $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha X \beta$, $\alpha, \beta \in V^*$;
- $A \in V_N$ is **productive** if there exists a derivation $A \overset{*}{\underset{G}{\Longrightarrow}} u$, with $u \in V_T^*$ (the other nonterminal symbols are called **useless**);
- G is a **reduced grammar** if all symbols are accessible and all nonterminal symbols are productive;
- $A \in V_N$ is **left-recursive**, if there exists a derivation $A \overset{+}{\underset{G}{\Longrightarrow}} A\,\alpha,\ \alpha \in V^+$;
- $A \in V_N$ is **right-recursive**, if there exists a derivation $A \overset{+}{\underset{G}{\Longrightarrow}} \beta\,A$, where $\beta \in V^+$;
- G is **left (right) recursive grammar** if there exists a left (right) recursive symbol $A \in V_N$;
- **the set of sentential forms of the grammar** G**:** $S(G) = \{\alpha \in V^* \mid \exists\ S \overset{*}{\underset{G}{\Longrightarrow}} \alpha\}$.
- **the language of the grammar** G**:** $L(G) = \{w \in V_T^* \mid \exists\ S \overset{*}{\underset{G}{\Longrightarrow}} w\}$ (in fact, $L(G) = S(G) \cap V_T^*$);
- if $\alpha = \alpha_1\,\alpha_2 \ldots \alpha_k$ is a word over V, $\alpha_i \in V$, then $\widetilde{\alpha} = \alpha_k \ldots \alpha_2\,\alpha_1$ is called **the reverse (mirror) of** α;
- let $G = (V_N, V_T, S, P)$ be a context-free grammar. Then we denote by $\widetilde{G} = (V_N, V_T, S, \widetilde{P})$, where $\widetilde{P} = \{A \to \widetilde{\beta} \mid A \to \beta \in P\}$, its **reverse (mirror) grammar**.

Notations:

- nonterminal symbols: S (start symbol), A, B, ...
- terminal symbols: a, b, c, ...
- symbols (terminal or nonterminal): X, Y, ...
- terminal words: u, v, x, y, w, ...
- words (over terminal and nonterminal symbols): α, β, γ ...

- derivations: $\overset{r}{\underset{G}{\Longrightarrow}}$ means that the production r was applied in G; $\overset{\pi}{\underset{G}{\Longrightarrow}}$ refers to a sequence of productions (syntactic analysis); $\overset{0}{\underset{G}{\Longrightarrow}}$ means that no production has been applied;
- let $w = w_1\, w_2 \ldots w_k$ be a word over V. Then
  - ${}^{(m)}w = \begin{cases} w_{k-m+1}\, w_{k-m+2} \ldots w_k & \text{if } m \leq k \\ w & \text{otherwise} \end{cases}$
  - $w^{(n)} = \begin{cases} w_1\, w_2 \ldots w_n & \text{if } n \leq k \\ w & \text{otherwise} \end{cases}$
- $\mathbf{N}$ ($\mathbf{N}^+$ respectively) denotes the set of (strict positive) natural numbers.

We shall define a device similar to a nondeterministic pushdown "transducer". This will be called the **general bidirectional parser** attached to the context free grammar G. It scans two "input characters" (or strings) from left to right or right to left. It can push or pop strings in two stacks. The output tapes provide the syntactical analysis. It returns the value "ACC" or "REJ" depending on whether the input string is accepted or not.

In Section 4, we shall see how a deterministic bidirectional parsers can be designed (for some subclasses of context free languages).

Let G be an arbitrary context free grammar and $\widetilde{G}$ its reverse. Depending on how we can visit the derivation tree associated to its frontier, labelled by $w \in V_T^*$, we distinguish two strategies:

(a) descendant left to right strategy for G and right to left ascendant strategy for $\widetilde{G}$;

(b) ascendant left to right strategy for G and right to left descendant strategy for $\widetilde{G}$.

These two strategies can be depicted as:

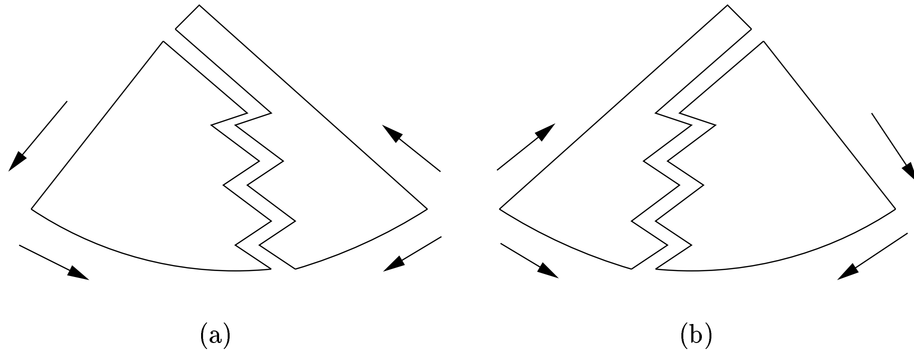


Figure 1. Strategies for bidirectional parsing

In the following, we shall present in a formal manner only the situation (a).

Definition 1.1 *Let $G=(V_N,V_T,S,P)$ be a context free grammar. Let $\mathcal{C}\subseteq\{s_1,s_2\}\times\{1,2,...,|P|\}^*\times V^*\#\times\#V_T^*\#\times V^*\#\times\{1,2,...,|P|\}^*\cup\{ACC,REJ\}$ be the set of all possible configurations, where $\#$ is a special character (a new terminal symbol). The* **general left bidirectional parser** *(denoted by $GBP_l(G)$) is the pair $(\mathcal{C}_0,\vdash)$, where $\mathcal{C}_0=\{(s_1,\lambda,S\#,\#w\#,\#,\lambda)\mid w\in V_T^*\}\subseteq\mathcal{C}$ is called the set of* **initial configurations**. *The first component is the state, the second and the last components of a configuration are for storing the partial syntactic analysis. The third and the fifth components are the work - stacks (each of which has at the bottom the marker $\#$). The fourth component represents the current content of the input word (with the two markers). The* **transition relation** *($\vdash\subseteq\mathcal{C}\times\mathcal{C}$, sometimes denoted by $\underset{GBP_l(G)}{\longmapsto}$) between configurations is given by:*

1^0 *Expand-Shift:* $(s_1,\pi_1,A\,\alpha\#,\#u\,b\#,\beta\#,\pi_2)\vdash(s_1,\pi_1 r,\delta\alpha\#,\#u\#,b\beta\#,\pi_2)$, *where* $r=A\to\delta\in P$;

2^0 *Expand-Reduce:* $(s_1,\pi_1,A\alpha\#,\#u\#,\varepsilon\beta\#,\pi_2)\vdash(s_1,\pi_1 r_1,\delta\alpha\#,\#u\#,B\beta\#,$ $r_2\pi_2)$, *where* $r_1=A\to\delta\in P$ *and* $r_2=B\to\varepsilon\in P$;

3^0 *Reduce-Shift:* $(s_1,\pi_1,a\,\alpha\#,\#a\,u\,b\#,\beta\#,\pi_2)\vdash(s_1,\pi_1,\alpha\#,\#u\#,b\,\beta\#,\pi_2)$;

4^0 *Reduce-Reduce:* $(s_1,\pi_1,a\,\alpha\#,\#a\,u\#,\varepsilon\,\beta\#,\pi_2)\vdash(s_1,\pi_1,\alpha\#,\#u\#,B\,\beta\#,$ $r\pi_2)$, *where* $r=B\to\varepsilon\in P$;

5^0 *Expand-Stay:* $(s_1,\pi_1,A\,\alpha\#,\#u\#,\beta\#,\pi_2)\vdash(s_1,\pi_1 r,\delta\,\alpha\#,\#u\#,\beta\#,\pi_2)$, *where* $r=A\to\delta\in P$;

6^0 *Reduce-Stay:* $(s_1,\pi_1,a\,\alpha\#,\#a\,u\#,\beta\#,\pi_2)\vdash(s_1,\pi_1,\alpha\#,\#u\#,\beta\#,\pi_2)$;

7^0 *Stay-Shift:* $(s_1,\pi_1,\alpha\#,\#u\,b\#,\beta\#,\pi_2)\vdash(s_1,\pi_1,\alpha\#,\#u\#,b\,\beta\#,\pi_2)$;

8^0 *Stay-Reduce:* $(s_1,\pi_1,\alpha\#,\#u\#,\varepsilon\,\beta\#,\pi_2)\vdash(s_1,\pi_1,\alpha\#,\#u\#,B\,\beta\#,r\pi_2)$, *where* $r=B\to\varepsilon\in P$;

9^0 *Possible-accept:* $(s_1,\pi_1,\gamma_1\#,\#\#,\gamma_2\#,\pi_2)\vdash(s_2,\pi_1,\gamma_1\#,\#\#,\gamma_2\#,\pi_2)$;

10^0 *Parallel-reduce:* $(s_2,\pi_1,X\gamma_1\#,\#\#,X\gamma_2\#,\pi_2)\vdash(s_2,\pi_1,\gamma_1\#,\#\#,\gamma_2\#,\pi_2)$;

11^0 *Accept:* $(s_2,\pi_1,\#,\#\#,\#,\pi_2)\vdash ACC$;

12^0 *Reject:* $(s_1,\pi_1,\alpha\#,\#u\#,\beta\#,\pi_2)\vdash REJ$ *and* $(s_2,\pi_1,\alpha\#,\#\#,\beta\#,\pi_2)\vdash REJ$ *if no transitions of type* 1^0, 2^0, ..., 11^0 *can be applied.*

The *deterministic two-stack machine* ([HoU79]), which is a deterministic Turing machine with a read-only input and two storage tapes is known to have the same power as the usual Turing machines. If a head moves left on either tape, a blank is printed on that tape.

"Lemma 7.3. ([HoU79]) *An arbitrary single-tape Turing machine can be simulated by a deterministic two-stack machine.*"

Another model equivalent to Turing machines is the two-counter machine, which are off-line Turing machines whose storage is semi-infinite, and whose alphabets contain only two symbols, Z and B (blank). Furthermore, the symbol Z, which serves as a bottom of stack, occurs initially on the cell scanned by the tape head and may never appear on any other cell. An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. It can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal.

"Theorem 7.9. ([HoU79]) *A two-counter machine can simulate an arbitrary Turing machine.*"

Our model is in fact a two-stack machine, the differences consist in the existence of two heads (instead of only one) which may read the input tape, and two output tapes which can be accessed only in write style, and has only two states. Therefore, our model can simulate a Turing machine.

This model can be depicted in the following figure:

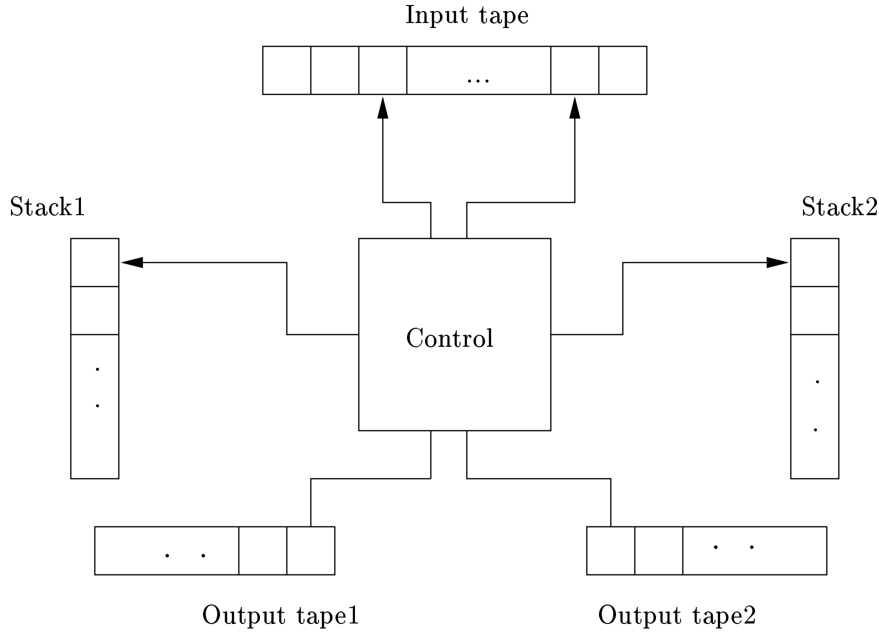


Figure 2. General Bidirectional Parser Style

Lemma 1.1 *Let G be a context free grammar. If*

$$(1) \qquad (s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \overset{*}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$$

then $S \overset{\pi_1}{\underset{G,lm}{\Longrightarrow}} u_1\, \alpha$ *and* $\beta \overset{\pi_2}{\underset{G,lm}{\Longrightarrow}} u_3$.

Proof We proceed by induction on the number of transitions (denoted by t). As some notations, $\vdash^{t,*}_{GBP_l(G)}$ and $\vdash^{1^0}_{GBP_l(G)}$, mean that t transitions, respectively the transition 1^0 have been applied.

Basis: $t = 1$. Starting from the initial configuration, we can apply transitions 1^0, 5^0 and 7^0. Supposing that we apply 1^0, we obtain:

$$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \vdash^{1^0}_{GBP_l(G)} (s_1, r, \alpha\#, \#u_1\, u_2\, u_3'\#, a\#, \lambda),$$

where $r = S \to \alpha \in P$ and $u_3 = u_3'\, a$. Therefore $S \overset{r}{\underset{G,lm}{\Longrightarrow}} \alpha$ and $a \overset{0}{\underset{G,lm}{\Longrightarrow}} a$. The other cases ($5^0$, 7^0) can be shown in a similar way.

Inductive Step: We suppose that the relation (1) is true for at most t transitions and prove it for $t + 1$ transitions. We know that:

$$(2) \qquad (s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \vdash^{t+1,*}_{GBP_l(G)} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$$

We have to prove that $S \overset{\pi_1}{\underset{G,lm}{\Longrightarrow}} u_1\, \alpha$ and $\beta \overset{\pi_2}{\underset{G,lm}{\Longrightarrow}} u_3$. The last transition in (2) may be of the type 1^0, 2^0, ..., 8^0.

I: We suppose that the last transition in (2) is of the form *expand-shift*. Now, we rewrite (2) into:

$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \vdash^{t,*}_{GBP_l(G)} (s_1, \pi_1', A\alpha_2\#, \#u_2\, b\#, \beta'\#, \pi_2)$

$\vdash^{1^0}_{GBP_l(G)} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$, where $r = A \to \alpha_1 \in P$, $\pi' r = \pi_1$, $\alpha_1\, \alpha_2 = \alpha$, $b\, \beta' = \beta$. According to the inductive hypothesis, we obtain:

$$S \overset{\pi_1'}{\underset{G,lm}{\Longrightarrow}} u_1\, A\, \alpha_2 \text{ and } b\, \beta' = \beta \overset{\pi_2}{\underset{G,lm}{\Longrightarrow}} u_3.$$

But $r = A \to \alpha_1$ is a production from P, so

$$S \overset{\pi_1'}{\underset{G,lm}{\Longrightarrow}} u_1\, A\, \alpha_2 \overset{r}{\underset{G,lm}{\Longrightarrow}} u_1\, \alpha_1\, \alpha_2 = u_1\, \alpha.$$

II: We suppose that the last transition in (2) is of the form *expand-reduce*. Now, we can rewrite the initial transition:

$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \vdash^{t,*}_{GBP_l(G)} (s_1, \pi_1', A\, \alpha_2\#, \#u_2\#, \varepsilon\, \beta'\#, \pi_2)$

$\vdash^{2^0}_{GBP_l(G)} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $r_1 = A \to \alpha_1 \in P$, $\pi_1' r_1 = \pi_1$, $\alpha_1\, \alpha_2 = \alpha$, $r_2 = B \to \varepsilon \in P$, $B\, \beta' = \beta$, $r_2 \pi_2' = \pi_2$. According to the inductive hypothesis, we obtain:

$$S \overset{\pi_1'}{\underset{G,lm}{\Longrightarrow}} u_1\, A\, \alpha_2 \text{ and } \varepsilon\, \beta' \overset{\pi_2'}{\underset{G,lm}{\Longrightarrow}} u_3.$$

But $r_1 = A \to \alpha_1$ and $r_2 = B \to \varepsilon$ are productions from P, so

$$S \underset{G,lm}{\overset{\pi_1'}{\Longrightarrow}} u_1\, A\, \alpha_2 \underset{G,lm}{\overset{r_1}{\Longrightarrow}} u_1\, \alpha_1\, \alpha_2 = u_1\, \alpha \text{ and } \beta = B\beta' \underset{G,lm}{\overset{r_2}{\Longrightarrow}} \varepsilon\, \beta' \underset{G,lm}{\overset{\pi_2'}{\Longrightarrow}} u_3$$

III: We suppose that the last transition in (2) is of the form *reduce-shift*. Now, we can rewrite the initial transition:
$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \underset{GBP_l(G)}{\overset{t,*}{\longmapsto}} (s_1, \pi_1', a\, \alpha_2\#, \#a\, u_2\, b\#, \beta'\#, \pi_2)$
$\underset{GBP_l(G)}{\overset{3^0}{\longmapsto}} (\pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $b\, \beta' = \beta$. According to the inductive hypothesis, we obtain:

$$S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1'\, a\, \alpha \text{ and } \beta' \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} u_3'$$

where $u_1'\, a = u_1$ and $b\, u_3' = u_3$.
Therefore

$$S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1\, a\, \alpha = u_1\, \alpha \text{ and } \beta = b\beta' \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} b\, u_3' = u_3.$$

IV: We suppose that the last transition in (2) is of the form *reduce-reduce*. The initial transition becomes:
$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \underset{GBP_l(G)}{\overset{t,*}{\longmapsto}} (s_1, \pi_1, a\, \alpha\#, \#a\, u_2\#, \delta\, \beta'\#, \pi_2')$
$\underset{GBP_l(G)}{\overset{4^0}{\longmapsto}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $r = B \to \delta \in P$, $B\, \beta' = \beta$, $r\pi_2' = \pi_2$.
According to the inductive hypothesis, we obtain:

$$S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1\, a\, \alpha \text{ and } \delta\, \beta' \underset{G,lm}{\overset{\pi_2'}{\Longrightarrow}} u_3.$$

where $u_1'\, a = u_1$.
Therefore

$$S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1\, a\, \alpha = u_1\, \alpha \text{ and } \beta = B\beta' \underset{G,lm}{\overset{r}{\Longrightarrow}} \delta\, \beta' \underset{G,lm}{\overset{\pi_2'}{\Longrightarrow}} u_3$$

The other cases *expand-stay*, *reduce-stay*, *stay-shift*, *stay-reduce* are similar to the first four transitions. For instance, the case *expand - stay* can be solved in this way:

- the *expand* transitions can be treated in the same manner as the situations I and II;
- the *stay* transitions can be solved directly from the inductive hypothesis.

■

Lemma 1.2 *Let G be a context free grammar. If $S \overset{\pi_1}{\underset{G,lm}{\Longrightarrow}} u_1\,\alpha$ and $\beta \overset{\pi_2}{\underset{G,lm}{\Longrightarrow}} u_3$, where* [1] *$\alpha \in V_N$ or $\alpha = \lambda$, then*

$$(s_1, \lambda, S\#, \#u_1\,u_2\,u_3\#, \#, \lambda) \overset{*}{\underset{GBP_l(G)}{\longmapsto}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$$

Proof By induction on $t = |\pi_1| + |\pi_2|$.

Basis: We present two initial cases ($t = 0$ and $t = 1$).

- $|\pi_1| = 0$ and $|\pi_2| = 0$. Then the hypothesis may be written as $S \overset{0}{\underset{G,lm}{\Longrightarrow}} S$, $u_3 \overset{0}{\underset{G,lm}{\Longrightarrow}} u_3$ and $\beta = u_3$. Therefore $\alpha = S$ and $u_1 = \lambda$. Applying $|u_3|$ *stay - shift* transitions starting from the initial configuration, we obtain:

$$(s_1, \lambda, S\#, \#u_1\,u_2\,u_3\#, \#, \lambda) \overset{7^0,*}{\underset{GBP_l(G)}{\longmapsto}} (s_1, \lambda, \alpha\#, \#u_2, \beta\#, \#, \lambda)$$

- $|\pi_1| = 1$ and $|\pi_2| = 0$. Then the hypothesis may be rewritten in $S \overset{\pi}{\underset{G,lm}{\Longrightarrow}} u_1\,\alpha$ and $u_3 \overset{0}{\underset{G,lm}{\Longrightarrow}} u_3$. Therefore $u_3 = \beta$. Applying an *expand - stay* transition, starting from the initial configuration, we obtain:

$$(s_1, \lambda, S\#, \#u_1\,u_2\,u_3\#, \#, \lambda) \overset{5^0}{\underset{GBP_l(G)}{\longmapsto}} (s_1, r, u_1\,\alpha\#, \#u_1\,u_2\,u_3\,\#, \#, \lambda)$$

  Now, we distinguish two cases:

  - $|u_1| \geq |u_3|$. Then we continue with $|u_1|$ transitions of the form *reduce - shift* obtaining the configuration:

  $$(s_1, r, \alpha\#, \#u_2\,u_3'\#, u_3''\#, \lambda), \text{ where } u_3'\,u_3'' = u_3.$$

    Now, we apply $|u_3'|$ transitions of the form *stay - shift* and we obtain the final configuration:

  $$(s_1, r, \alpha\#, \#u_2\#, \beta\#, \lambda);$$

  - $|u_1| < |u_3|$. Then we continue with $|u_3|$ transitions of the form *reduce - shift* obtaining the configuration:

  $$(s_1, r, u_1'\,\alpha\#, \#u_1'\,u_2\#, u_3\#, \lambda).$$

    Now, we apply $|u_1'|$ transitions of the form *reduce - stay* and we obtain the final configuration:

  $$(s_1, r, \alpha\#, \#u_2\#, \beta\#, \lambda).$$

- $|\pi_1| = 0$ and $|\pi_2| = 1$. Then the hypothesis may be written as $S \underset{G,lm}{\overset{0}{\Longrightarrow}} S$ and $\beta \underset{G,lm}{\overset{r}{\Longrightarrow}} u_3$. Therefore $\alpha = S$, $u_1 = \lambda$, $\beta \in V_N$ and $u_3 \in V_T^*$. It follows that we have to apply $|u_3|$ transitions of type *stay - shift* obtaining:

$$(s_1, \lambda, S\#, \#u_2\#, u_3\#, \lambda).$$

Now, we apply a transition of type *stay - reduce*, obtaining the final configuration:

$$(s_1, \lambda, \alpha\#, \#u_2\#, \beta\#, r).$$

Inductive Step: We have to prove that $P(t) \to P(t+1)$, where P is a logic predicat equivalent to our implication. This means that we distinguish two cases (I: $\pi_1 = \pi_1' \, r_1$ and $\pi_2 = \pi_2'$) and (II: $\pi_1 = \pi_1'$ and $\pi_2 = r_2 \, \pi_2'$).

I: Let $r_1 = A \to \delta$ be the last applied production in the derivation $S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1 \, \alpha$. We have

$$S \underset{G,lm}{\overset{\pi_1'}{\Longrightarrow}} u_1' \, \alpha' = u_1' \, A \, \alpha_1' \underset{G,lm}{\overset{r_1}{\Longrightarrow}} u_1' \, \delta \, \alpha_1' = u_1 \, \alpha,$$

where $\delta = u_1'' \, \alpha_1''$, $\alpha_1'' \, \alpha_1' = \alpha$, $u_1' \, u_1'' = u_1$. Because $^{(1)}\alpha' \in V_N$, we can apply the inductive hypothesis:

$$(s_1, \lambda, S\#, \#u_1 \, u_2 \, u_3\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1', \alpha'\#, \#u_1'' \, u_2\#, \beta\#, \pi_2') =$$

$$= (s_1, \pi_1', A \, \alpha_1'\#, \#u_1'' \, u_2\#, \beta\#, \pi_2') \underset{GBP_l(G)}{\overset{5^0}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1' \, r_1, \delta \, \alpha_1'\#, \#u_1'' \, u_2\#, \beta\#, \pi_2') =$$

$$= (s_1, \pi_1, u_1'' \, \alpha_1'' \, \alpha_1'\#, \#u_1'' \, u_2\#, \beta\#, \pi_2').$$

Now, by applying $|u_1''|$ transitions of type 6^0, we obtain the configuration (because $\alpha_1'' \, \alpha_1' = \alpha$ and $\pi_2' = \pi_2$):

$$(s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2).$$

II: Let $r_2 = B \to \varepsilon$ be the last applied production in the derivation $\beta \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} u_3$. Therefore, we have

$$\beta = u_3' \, B \, \beta' \underset{G,lm}{\overset{r_2}{\Longrightarrow}} u_3' \, \varepsilon \, \beta' \underset{G,lm}{\overset{\pi_2'}{\Longrightarrow}} u_3$$

So, because $u_3 = u_3' \, u_3''$, we can consider the derivation $\varepsilon \, \beta' \underset{G,lm}{\overset{\pi_2'}{\Longrightarrow}} u_3''$. Now, applying the inductive hypothesis, we get:

$$(s_1, \lambda, S\#, \#u_1 \, u_2 \, u_3\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1', \alpha\#, \#u_2 \, u_3'\#, \varepsilon \, \beta'\#, \pi_2')$$

$\underset{GBP_l(G)}{\overset{8^0}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1', \alpha\#, \#u_2 \, u_3'\#, B \, \beta'\#, r_2 \, \pi_2')$. Continuing with $|u_3'|$ transitions of type 7^0, we obtain the configuration ($\pi_1' = \pi_1$, $r_2 \, \pi_2' = \pi_2$ and $u_3' \, B \, \beta' = \beta$):

$$(s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2).$$

■

Theorem 1.1 *Let G be a context free grammar. Then*

a) $(s_1, \lambda, S\#, \#w\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash}} (s_1, \pi_1, \gamma\#, \#\#, \gamma\#, \pi_2)$ *iff* $S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} \gamma \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} w$;

b) $(s_1, \lambda, S\#, \#w\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash}} (s_2, \pi_1, \#, \#\#, \#, \pi_2) \underset{GBP_l(G)}{\overset{11^0}{\vdash}} ACC$ *iff* $S \underset{G,lm}{\overset{\pi_1\pi_2}{\Longrightarrow}} w$.

Proof

a) Taking in Lemmas 1.1 and 1.2, $u_1 = u_2 = \lambda$, $u_3 = w$, $\alpha = \beta = \gamma$.
b) The only transition for which we can obtain ACC is 11^0.

($\Longrightarrow$) Because from state s_2 it is impossible to return to state s_1, it follows that there exists a $\gamma \in V^*$ for which $(s_1, \lambda, S\#, \#w\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash}} (s_2, \pi_1, \gamma\#, \#\#, \gamma\#, \pi_2)$. According to a), it follows that $S \underset{G,lm}{\overset{\pi_1\pi_2}{\Longrightarrow}} w$.

($\Longleftarrow$) Since $S \underset{G,lm}{\overset{\pi_1\pi_2}{\Longrightarrow}} w$, it follows that there exists a $\gamma \in V^*$ for which

$$S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} \gamma \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} w.$$

According to a), we get

$$(s_1, \lambda, S\#, \#w\#, \#, \lambda) \underset{GBP_l(G)}{\overset{*}{\vdash}} (s_1, \pi_1, \gamma\#, \#\#, \gamma\#, \pi_2).$$

Now, applying transition 9^0, followed by $|\gamma|$ transitions of type 10^0, and finally by 11^0, we obtain the conclusion. ■

For describing the situation (b) from Figure 1, we just "reverse" the general left bidirectional parser (by switching the first two components with the last two components from the configurations).

Definition 1.2 *Let $G = (V_N, V_T, S, P)$ be a context free grammar. Let $\mathcal{C} \subseteq \{s_1, s_2\} \times \{1, 2, ..., |P|\}^* \times \#V^* \times \#V_T^*\# \times \#V^* \times \{1, 2, ..., |P|\}^* \cup \{ACC, REJ\}$ be the set of all possible configurations, where $\#$ is a special character (a new terminal symbol). The* **general right bidirectional parser** *($GBP_r(G)$) is the pair $(\mathcal{C}_0, \vdash)$, where $\mathcal{C}_0 = \{(\lambda, \#, \#w\#, \#S, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**, *and $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ is the* **transition relation** *(sometimes denoted by $\underset{GBP_r(G)}{\vdash}$) between configurations given by:*

1^0 *Shift-Expand:* $(s_1, \pi_1, \#\beta, \#bu\#, \#\alpha A, \pi_2) \vdash (s_1, \pi_1, \#\beta b, \#u\#, \#\alpha\delta, \pi_2 r)$, *where* $r = A \to \delta \in P$;

2^0 *Reduce-Expand:* $(s_1, \pi_1, \#\beta\varepsilon, \#u\#, \#\alpha A, \pi_2) \vdash (s_1, r_1\pi_1, \#\beta\, B, \#u\#, \#\alpha\, \delta, \pi_2 r_2)$, *where* $r_1 = B \to \varepsilon \in P$ *and* $r_2 = A \to \delta \in P$;

3^0 *Shift-Reduce:* $(s_1, \pi_1, \#\beta, \#b\, u\, a\#, \#\alpha\, a, \pi_2) \vdash (s_1, \pi_1, \#\beta\, b, \#u\#, \#\alpha, \pi_2)$;

4^0 *Reduce-Reduce:* $(s_1, \pi_1, \#\beta\varepsilon, \#u\, a\#, \#\alpha\, a, \pi_2) \vdash (s_1, r\pi_1, \#\beta\, B, \#u\#, \#\alpha, \pi_2)$, *where* $r = B \to \varepsilon \in P$;

5^0 *Shift-Stay:* $(s_1, \pi_1, \#\beta, \#b\, u\#, \#\alpha, \pi_2) \vdash (s_1, \pi_1, \#\beta\, b, \#u\#, \#\alpha, \pi_2)$;

6^0 *Reduce-Stay:* $(s_1, \pi_1, \#\beta\, \varepsilon, \#u\#, \#\alpha, \pi_2) \vdash (s_1, r\pi_1, \#\beta\, B, \#u\#, \#\alpha, \pi_2)$;

7^0 *Stay-Expand:* $(s_1, \pi_1, \#\beta, \#u\#, \#\alpha\, A, \pi_2) \vdash (s_1, \pi_1, \#\beta, \#u\#, \#\alpha\, \delta, \pi_2 r)$, *where* $r = A \to \delta \in P$;

8^0 *Stay-Reduce:* $(s_1, \pi_1, \#\beta, \#u\, a\#, \#\alpha\, a, \pi_2) \vdash (s_1, \pi_1, \#\beta, \#u\#, \#\alpha, \pi_2)$;

9^0 *Possible-accept:* $(s_1, \pi_1, \#\gamma_1, \#\#, \#\gamma_2, \pi_2) \vdash (s_2, \pi_1, \#\gamma_1, \#\#, \#\gamma_2, \pi_2)$;

10^0 *Parallel-reduce:* $(s_2, \pi_1, \#\gamma_1 X, \#\#, \#\gamma_2 X, \pi_2) \vdash (s_2, \pi_1, \#\gamma_1, \#\#, \#\gamma_2, \pi_2)$;

11^0 *Accept:* $(s_2, \pi_1, \#, \#\#, \#, \pi_2) \vdash ACC$;

12^0 *Reject:* $(\{s_1, s_2\}, \pi_1, \#\alpha, \#u\#, \#\beta, \pi_2) \vdash REJ$ *if no transitions of type* 1^0, 2^0, ..., 11^0 *can be applied.*

We shall show the correctness of the right general bidirectional parser.

Lemma 1.3 *Let G be a context free grammar. If*

$$(3) \qquad (s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#S, \lambda) \overset{*}{\underset{GBP_r(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \#\beta, \#u_2\#, \#\alpha, \pi_2)$$

then $\beta \overset{\pi_1}{\underset{G,rm}{\Longrightarrow}} u_1$ *and* $S \overset{\pi_2}{\underset{G,rm}{\Longrightarrow}} \alpha\, u_3$.

Proof Analogous to the proof of Lemma 1.1, we can proceed by induction on the number of transitions. ■

Lemma 1.4 *Let G be a context free grammar. If* $\beta \overset{\pi_1}{\underset{G,rm}{\Longrightarrow}} u_1$ *and* $S \overset{\pi_2}{\underset{G,rm}{\Longrightarrow}} \alpha\, u_3$, *where* $\alpha^{(1)} \in V_N$ *or* $\alpha = \lambda$, *then*

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#S, \lambda) \overset{*}{\underset{GBP_r(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \#\beta, \#u_2\#, \#\alpha, \pi_2).$$

Proof Analogous to the proof of Lemma 1.2, by induction on $|\pi_1| + |\pi_2|$. ■

Theorem 1.2 *Let G be a context free grammar. Then*

a) $(s_1, \lambda, \#, \#w\#, \#S, \lambda) \overset{*}{\underset{GBP_r(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \#\gamma, \#\#, \#\gamma, \pi_2)$ *iff* $S \overset{\pi_2}{\underset{G,rm}{\Longrightarrow}} \gamma \overset{\pi_1}{\underset{G,rm}{\Longrightarrow}} w$;

b) $(s_1, \lambda, \#, \#w\#, \#S, \lambda) \overset{*}{\underset{GBP_r(G)}{\vdash\!\!\!-\!\!\!-}} ACC$ *iff* $S \overset{*}{\underset{G,rm}{\Longrightarrow}} w$.

Proof Taking in Lemmas 1.3 and 1.4, $u_1 = w$, $u_2 = u_3 = \lambda$, $\alpha = \beta = \gamma$. ■

Example 1.1 *Let us consider the context free grammar given by the productions* $1.\, A \to a\,A\,B\,b$ $2.\, A \to c\,d$ $3.\, B \to B\,C\,e$ $4.\, B \to f$ $5.\, C \to g$*, and the word* $w = a\,c\,d\,f\,g\,e\,b$*. We shall see how the* $GBP_l(G)$ *and* $GBP_r(G)$ *can work having as input the word* w *(we shall write on the sign* $\vdash$ *the associated transition number).*

For $GBP_l(G)$ *we obtain:*

$$(s_1, \lambda, S\#, \#a\,c\,d\,f\,g\,e\,b\#, \#, \lambda) \overset{1^0}{\vdash} (s_1, [1], a\,A\,B\,b\#, \#a\,c\,d\,f\,g\,e\#, b\#, \lambda) \overset{3^0}{\vdash}$$
$$(s_1, [1], A\,B\,b\#, \#c\,d\,f\,g\#, e\,b\#, \lambda) \overset{1^0}{\vdash} (s_1, [1,2], c\,d\,B\,b\#, \#c\,d\,f\#, g\,e\,b\#, \lambda) \overset{4^0}{\vdash}$$
$$(s_1, [1,2], d\,B\,b\#, \#d\,f\#, C\,e\,b\#, [5]) \overset{3^0}{\vdash} (s_1, [1,2], B\,b\#, \#\#, f\,C\,e\,b\#, [5]) \overset{8^0}{\vdash}$$
$$(s_1, [1,2], B\,b\#, \#\#, B\,C\,e\,b\#, [4,5]) \overset{8^0}{\vdash} (s_1, [1,2], B\,b\#, \#\#, B\,b\#, [3,4,5]) \overset{9^0}{\vdash}$$
$$(s_2, [1,2], B\,b\#, \#\#, B\,b\#, [3,4,5]) \overset{10^0}{\vdash} (s_2, [1,2], b\#, \#\#, b\#, [3,4,5]) \overset{10^0}{\vdash}$$
$$(s_2, [1,2], \#, \#\#, \#, [3,4,5]) \overset{11^0}{\vdash} ACC$$

Denoting by π_{lm} *the left most derivation obtained by concatenating the lists* $[1,2]$ *and* $[3,4,5]$*, we can conclude that* $w \in L(G)$ *and* $S \underset{G}{\overset{\pi_{lm}}{\Longrightarrow}} w$*.*

For $GBP_r(G)$ *we obtain:*

$$(s_1, \lambda, \#, \#a\,c\,d\,f\,g\,e\,b\#, \#S, \lambda) \overset{1^0}{\vdash} (s_1, \lambda, \#a, \#c\,d\,f\,g\,e\,b\#, \#a\,A\,B\,b, [1]) \overset{3^0}{\vdash}$$
$$(s_1, \lambda, \#a\,c, \#d\,f\,g\,e\#, \#a\,A\,B, [1]) \overset{1^0}{\vdash} (s_1, \lambda, \#acd, \#fge\#, \#aABCe, [1,3]) \overset{4^0}{\vdash}$$
$$(s_1, [2], \#aA, \#fg\#, \#aABC, [1,3]) \overset{1^0}{\vdash} (s_1, [2], \#aAf, \#g\#, \#aABg, [1,3,5]) \overset{4^0}{\vdash}$$
$$(s_1, [2,4], \#aAB, \#\#, \#aAB, [1,3,5]) \overset{9^0}{\vdash} (s_2, [2,4], \#aAB, \#\#, \#aAB, [1,3,5])$$
$$\overset{10^0}{\vdash} (s_2, [2,4], \#aA, \#\#, \#aA, [1,3,5]) \overset{10^0}{\vdash} (s_2, [2,4], \#a, \#\#, \#a, [1,3,5]) \overset{10^0}{\vdash}$$
$$(s_2, [2,4], \#, \#\#, \#, [1,3,5]) \overset{11^0}{\vdash} ACC$$

Denoting by π_{rm} *the right most derivation obtained by concatenating the lists* $[1,3,5]$ *and* $[2,4]$*, we can conclude that* $w \in L(G)$ *and* $S \underset{G}{\overset{\pi_{rm}}{\Longrightarrow}} w$*.*

It is obvious that the parsers $GBP_l(G)$ and $GBP_r(G)$ are nondeterministic. For example, if G would contain the production $C \to B\,C\,e$, then $GBP_l(G)$ has to backtrack in the following configuration $(s_1, [1,2], B\,\beta\#, \#\#, B\,C\,e\,b\#, [4,5])$.

The same we can say about $GBP_r(G)$. If G contains, for example, the production $C \to f$, then we need a backtracking step at the configuration $(s_1, [2], \#a\,A\,f, \#g\#, \#a\,A\,B\,g, [1,3,5])$.

We shall see in sections 3, 4 how we can avoid these backtracking steps by defining special subclasses of context free languages (for which we can present deterministic algorithms). Furthermore, these subclasses ($RL(k)$, $RR(k)$, SIP and the cartesian product combinations) are large enough for describing the behaviour for practical compilers.

Example 1.2 *For the sake of testing the power of the bidirectional parser model (Definitions 1.1 and 1.2), we shall allow the reading of two terminal symbols from the input tape (for the left part). Using this additional property, we shall design a* deterministic *bidirectional parser which can analyse the context sensitive language* $L = \{a^n\, b^n\, c^n \mid n \geq 1\}$. *In fact, we shall simulate the monotone grammar given by the following productions ([JuA97]):*

1. $A \to a\,A\,B\,c$ *2.* $A \to a\,b\,c$ *3.* $c\,B \to B\,c$ *4.* $b\,B \to b\,b$

As a initial configuration, we take $(A\#, \#w\#, \#)$, *where* $w \in \{a, b, c\}^*$ *is the input word. Assuming that the notations* u *and* α, β *stand for words (of any length) over* $\{a, b, c\}$, *and* $\{a, b, c, A, B\}$ *respectively, the transitions will be the following:*

1^0 $(A\,\alpha\#, \#a\,a\,u\#, \beta\#) \vdash (a\,A\,B\,c\,\alpha\#, \#a\,a\,u\#, \beta\#)$

2^0 $(A\,\alpha\#, \#a\,b\,u\#, \beta\#) \vdash (a\,b\,c\,\alpha\#, \#a\,b\,u\#, \beta\#)$

3^0 $(a\,\alpha\#, \#a\,u\,c\#, \beta\#) \vdash (\alpha\#, \#u\#, c\,\beta\#)$

4^0 $(b\,\alpha\#, \#b\,u\#, \beta\#) \vdash (\alpha\#, \#u\#, \beta\#)$

5^0 $(c\,\alpha\#, \#u\#, c\,\beta\#) \vdash (\alpha\#, \#u\#, \beta\#)$

6^0 $(B\,\alpha\#, \#b\,u\#, \beta\#) \vdash (b\,\alpha\#, \#u\#, \beta\#)$

7^0 $(\#, \#\#, \#) \vdash ACC$

8^0 $(\alpha\#, \#u\#, \beta\#) \vdash REJ$ *if no transitions of type* 1^0, *...,* 7^0 *can be applied.*

Obviously, the above bidirectional parser is deterministic because at each step at most one transition may be applied.

Theorem 1.3 *The following statement is fulfilled:*

$$(A\#, \#w\#, \#) \overset{*}{\vdash} ACC \text{ iff } \exists\, n \in \mathbf{N}_+,\ w = a^n\, b^n\, c^n \text{ such that } A \overset{*}{\Longrightarrow} w.$$

Proof

($\Longleftarrow$) We have to show that $(A\#, \#a^n\, b^n\, c^n\#, \#) \overset{*}{\vdash} ACC$. If $n = 1$ then

$(A\#, \#a^n\, b^n\, c^n\#, \#) \overset{2^0}{\vdash} (a\,b\,c\#, \#a\,b\,c\#, \#) \overset{3^0}{\vdash} (b\,c\#, \#b\#, c\#) \overset{4^0}{\vdash}$
$(c\#, \#\#, c\#) \overset{5^0}{\vdash} (\#, \#\#, \#) \overset{7^0}{\vdash} ACC$

If $n > 1$ we have

$(A\#, \#a^n\, b^n\, c^n\#, \#) \overset{1^0}{\vdash} (a\,A\,B\,c\#, \#a^n\, b^n\, c^n\#, \#) \overset{3^0}{\vdash}$
$(A\,B\,c\#, \#a^{n-1}\, b^n\, c^{n-1}\#, c\#) \overset{(1^0,3^0)^*}{\vdash} (A\,(B\,c)^{n-1}\#, \#a\,b^n\,c\#, c^{n-1}\#) \overset{2^0}{\vdash}$
$(a\,b\,c\,(B\,c)^{n-1}\#, \#a\,b^n\,c\#, c^{n-1}\#) \overset{3^0}{\vdash} (b\,c\,(B\,c)^{n-1}\#, \#b^n\#, c^n\#) \overset{4^0}{\vdash}$

$$(c\,(B\,c)^{n-1}\#, \#b^{n-1}\#, c^{n}\#) \overset{5^0}{\vdash} ((B\,c)^{n-1}\#, \#b^{n-1}\#, c^{n-1}\#) \overset{6^0}{\vdash}$$
$$(b\,c\,(B\,c)^{n-2}\#, \#b^{n-1}\#, c^{n-1}\#) \overset{(4^0,5^0,6^0)^*}{\longmapsto} (\#, \#\#, \#) \overset{7^0}{\vdash} ACC$$

($\Longrightarrow$) We know that $(A\#, \#w\#, \#) \overset{*}{\vdash} ACC$ and we have to show that there exists $n \in \mathbf{N}_+$, $w = a^n\,b^n\,c^n$ and $A \overset{*}{\Longrightarrow} w$.

From the initial configuration, we can apply only transitions of type 1^0 or 2^0.

If we apply 2^0, then $w = a\,b\,u$, and we get the configuration $(a\,b\,c\#, \#a\,b\,u\#, \#)$. We can apply only 3^0, i.e. $u = u_1\,c$. We get the configuration $(b\,c\#, \#b\,u'\#, c\#)$. Applying the transition 4^0, we obtain $(c\#, \#u'\#, c\#)$. The only possiblity to obtain ACC is to apply 5^0 and get $(\#, \#u'\#, \#)$ and from here it is obvious that $u' = \lambda$. Therefore $w = a\,b\,c$ $(n = 1)$.

If we apply 1^0, then $w = a\,a\,u$, and we obtain $(a\,A\,B\,c\#, \#a\,a\,u\#, \#)$. We can apply only 3^0, i.e. $u = u_1\,c$. We get the configuration $(A\,B\,c\#, \#a\,u_1\#, c\#)$. Now, again we can apply only transitions 1^0 and 2^0. But, if we apply 2^0, then we cannot apply 1^0 and 2^0 anymore. So, the general configuration will be (after $(n-2)$ applications of the transitions 1^0 and 3^0, $n \geq 2$):

$$(a\,b\,c\,(B\,c)^{n-1}\#, \#a\,u'\#, c^{n-1}\#),\ u_1 = a^{n-2}\,u'\,c^{n-2}$$

Because the only transition which can be applied is 3^0, it follows that $u' = u_1\,c$ and the next configuration is:

$$(b\,c\,(B\,c)^{n-1}\#, \#u'\#, c^{n}\#).$$

Now, because of the transition 4^0, it follows that $u_1' = b\,v_1$ and the next configuration is:

$$(c\,(B\,c)^{n-1}\#, \#v_1\#, c^{n}\#).$$

The only possiblity is to apply 5^0. We get:

$$((B\,c)^{n-1}\#, \#v_1\#, c^{n-1}\#).$$

Now, we can apply only 6^0. It follows the configuration:

$$(b\,c\,(B\,c)^{n-2}\#, \#v_1\#, c^{n-1}\#).$$

Continuing in the same manner as above, we finally obtain:

$$(\#, \#v_1'\#, \#),\ \text{where } v_1 = b^{n-1}\,v_1'.$$

Now, the only possibility to get ACC is to have $v_1' = \lambda$.

Then

$$w = a\,a\,u = a\,a\,u_1\,c = a\,a\,a^{n-2}\,u'\,c^{n-2}\,c = a^n\,u_1'\,c^n =$$
$$= a^n\,b\,v_1\,c^n = a^n\,b^n\,v_1'\,c^n = a^n\,b^n\,c^n\ (n \geq 2)$$

■

2 A parallel approach for (general) bidirectional parsing

In this section, we present a parallel approach which is very convenient for describing the general left and right bidirectional parsing strategies (Figure 3).

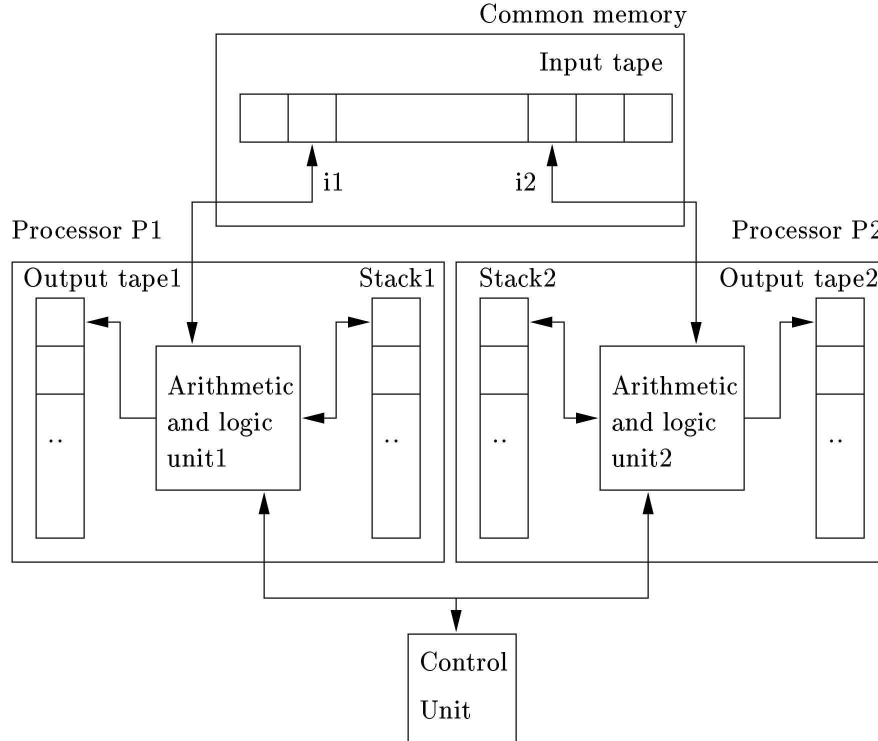


Figure 3. General SIMD Model for Left and Right Bidirectional Parsing

From the parallel architecture point of view ([Akl97]), we can say that our model is a SIMD (simple instruction stream and multiple data stream) computer. This means, in fact, that these two processors P1 and P2 operate under the control of a single instruction stream issued by a central control unit. The processors P1, P2 share a common memory.

Related to the analysis of parallel algorithms, the most important measure is *running time*, i.e. the time elapsed between the time when the first processor starts computing and the moment the last processor ends its computation. The running time of a parallel algorithm is usually obtained by counting two kinds of steps: *computational steps* and *routing steps*. A computational step is an arithmetic or logic operation performed on a datum within a processor. In a routing step, on the other hand, a datum travels from one processor to another via the shared memory or through the communication network. For a problem of size n, the parallel worst-case running time of an algorithm, a function of n,

will be denoted by $t(n)$. Strictly speaking, the running time is also a function of the number of processors. In general, computational steps and routing steps do not necessarily require the same number of time units. A routing step usually depends on the distance between the processors and typically takes a little longer to execute than a computational step.

To express lower and upper bounds of the number of steps required for solving a problem in the worst case, we present some notations. Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals:

(i) the function $g(n)$ is said to be *of order at least* $f(n)$, denoted $\Omega(f(n))$, if there are positive constants c and n_0 such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$;

(ii) the function $g(n)$ is said to be *of order at most* $f(n)$, denoted $\mathcal{O}(f(n))$, if there are positive constants c and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

For parallel algorithms, we take into consideration two additional factors (not only lower and upper bounds as in sequential case):

(i) the model of parallel computation used

(ii) the number of processors involved

In evaluating a parallel algorithm for a given problem, it is quite natural to do it in terms of the best available sequential algorithm for that problem. Thus a good indication of the quality of a parallel algorithm is the *speedup* it produces. This is defined as

$$Speedup = \frac{worst\!-\!case\ running\ time\ of\ fastest\ known\ sequential\ algorithm\ for\ the\ problem}{worst\!-\!case\ running\ time\ of\ parallel\ algorithm}$$

Clearly, the larger the speedup, the better the parallel algorithm.

The second most important criterion in evaluating a parallel algorithm is the **number of processors** it requires to solve a problem. It costs money to purchase, maintain, and run computers. When several processors are present, the problem of maintenance, in particular, is compound, and the price paid to guarantee a high degree of reliability rises sharply. Therefore, the larger the number of processors an algorithm uses to solve a problem, the more expensive the solution becomes to obtain. For a problem of size n, the number of processors required by an algorithm, a function of n, will be denoted by $p(n)$. Sometimes the number of processors is a constant independent of n.

The *cost* of a parallel algorithm is defined as

$$Cost = parallel\ running\ time \times number\ of\ processors\ used.$$

In other words, cost equals the number of steps executed simultaneously by all processors in solving a problem in the worst case. This definition assumes that

all processors execute the same number of steps. If it is not the case, then cost is an upper bound on the total number of steps executed. For a problem of size n, the cost of a parallel algorithm, will be denoted by $c(n)$ (i.e. $c(n) = t(n) \times p(n)$).

Assume that a lower bound on the number of sequential operations required in the worst case to solve a problem is known. If the cost of a parallel algorithm for that problem matches this lower bound to within a constant multiplicative factor, then the algorithm is said to be *cost optimal*. This is because any parallel algorithm can be simulated on a sequential computer. If the total number of steps executed during the simulation is equal to the lower bound, then this means that, when it comes to *cost*, this parallel algorithm cannot be improved upon as it executes the minimum number of steps possible. It may be possible, of course, to *reduce the running time* of a cost-optimal parallel algorithm by *using more processors*. Similarly, we may be able to *use fewer processors*, while retaining cost optimality, if we are willing to settle for a *higher running time*.

A parallel algorithm is *not cost optimal* if a sequential algorithm exists whose running time is smaller than the parallel algorithm's cost. Let $\Omega(T(n))$ be a lower bound on the number of sequential steps required to solve a problem of size n. Then $\Omega(T(n)/N)$ is a lower bound on the running time of any parallel algorithm that uses N processors to solve that problem.

When no optimal sequential algorithm is known for solving a problem, the *efficiency* of a parallel algorithm for that problem is used to evaluate its cost. This is defined as follows:

$$Efficiency = \frac{worst-case\ running\ time\ of\ fastest\ known\ sequential\ algorithm\ for\ the\ problem}{cost\ of\ parallel\ algorithm}$$

Usually, *efficiency* ≤ 1; otherwise a faster sequential algorithm can be obtained from the parallel one !

In the following, we shall present a parallel algorithm which describes the **general left bidirectional parsing strategy.** Our algorithm (denoted by (PAR_LEFT)) uses the following variables:

- `w` $\in V_T^*$ the input word; furthermore, `w` can be stored into local memories of that two processors; in that case, the only global variables (belonging to the common memory) are `i1` and `i2`;
- `n` the length of `w`;
- `i1`, `i2` two counters for the positions of the pointers to `w`;
- `is_no_over` a boolean variable which takes the `true` value if processor `P2` made a reduce action;
- `accept` a boolean variable which takes the `true` value iff $w \in L(G)$;
- `Stack1`, `Stack2` two working stacks for `P1` and `P2`;
- `Output_tape1`, `Output_tape2` the output tapes of `P1` and `P2` for storing the syntactic analysis of the input word;

- `exit` a boolean variable which is true iff `P1` or `P2` detect the non-acceptance of the input word.

Of course, we use some predefined procedures, such as:

- `pop(Stack,top)` - the value of `top` will be the value of the first symbol from `Stack`; after that the top of `Stack` will be removed;
- `push(Stack,`α`)` - push in the top of `Stack` the values $\alpha = \alpha_1\alpha_2...\alpha_k$ ($\alpha \in V^*,\ k = |\alpha|$); α_1 will be the new top of `Stack`;
- `push(Output_tape,r)` - push in the top of `Output_tape` the value of `r`.

Now, the method of parallel algorithm (PAR_LEFT) can be presented (we suppose that the context free grammar $G = (V_N, V_T, S, P)$ is already read):

```
begin
   read(n); read(w); i1:=1; i2:=n; push(Stack1, S);
   accept:=false; exit:=false;
   repeat in parallel
     if (i1<=i2) then action1(P1);
     action2(P2);
   until (i1>=i2) or (exit=true);
   if (i1>=i2) and (exit=false) then begin
     is_no_over:=false;
     while (is_no_over=false) and (exit=false) do
       if (Stack1=Stack2) then begin
         is_no_over:=true;
         accept:=true
       end
       else
         action2(P2)
   end;
   if (accept=true) then begin
     write('w is accepted and has the left syntactic analysis ');
     write(Output_tape1, Output_tape2);
   end
   else write('w is not accepted.');
end;
```

It remains to describe the procedures `action1(P1)` and `action2(P2)`.

```
procedure action1(P1);
begin
   pop(Stack1,top);
   case top of
     (top∈ V_T) and (top=w[i1]):
       /* reduce action */
```

```
      if (i1<i2) then i1:=i1+1;
    top∈ V_N: begin
      /* expand action */
      find a production r : top → α ∈ P;
      if (there exists an r of this form) then begin
        push(Stack1,α);
        push(Output_tape1,r);
      end
    end
    otherwise: begin
      backtrack step is needed;
      if (all the backtrack steps are over) and
        (still no reduce or expand action could be made)
      then exit:=true
    end
end;

procedure action2(P2);
begin
  case
    if (∃ r= B → β ∈ P, β is in Stack2 starting from top) then begin
      /* reduce action */
      pop(Stack2, β);
      push(Stack2, B);
      push(Output_tape2,r);
    end;
    if (i2>i1) then begin
      /* shift action */
      push(Stack2,w[i2]);
      i2:=i2-1;
    end;
    otherwise: begin
      backtrack step is needed;
      if (all the backtrack steps are over) and
        (still no reduce or expand action could be made)
      then exit:=true
    end
end;
```

Theorem 2.1 *(finiteness of the Algorithm (PAR_LEFT))*
The Algorithm (PAR_LEFT) performs a finite number of steps until terminates its execution.

Proof Because the input grammar G has a finite number of productions, the number of expand actions for `P1` and reduce actions for `P2` is finite. For the other actions (i.e. reduce for `P1` and shift for `P2`) we read one character from `w`. Thus, because `w` is a finite word, it follows that the statement `repeat-until` from the

Algorithm (PAR_LEFT) has a finite number of iterations (both processors `P1` and `P2` terminate their execution after a finite number of steps). Therefore, the Algorithm (PAR_LEFT) performs a finite number of steps until terminating its execution. ■

Theorem 2.2 *(correctness of the Algorithm (PAR_LEFT))*
For a given $G = (V_N, V_T, S, P)$ and $w \in V_T^*$ as its input, the Algorithm (PAR_LEFT) gives the answer 'w is accepted ...' if $w \in L(G)$ and 'w is not accepted.' otherwise.

Proof We present an informal proof, Algorithm (PAR_LEFT) being in fact a description in pseudocod of $GBP_l(G)$ according to Definition 1.1. More precisely, for showing the correctness of the parallel Algorithm (PAR_LEFT), it suffices to remark that (PAR_LEFT) is "equivalent" to the sequential algorithm associated to the transitions of the general left bidirectional parser. The only case which is not obviously true corresponds to the case $i1 = i2$ (only one letter from the input word remains to be read). Then P1 could make only expand actions, but P2 could make both operations, i.e. *reduce* and *shift* actions. This restriction was imposed for eliminating errors and the *deadlock*. Deadlock in that algorithm is understood in the following sense: both processors wait one each other to "read" the letter. Some errors refer to the fact that both processors "read" the letter (i.e. P1 makes `i1:=i1+1`, and P2 makes `i2:=i2-1`) and thus the content of that two stacks could not be equal, even for the words which belong to the language of the input grammar.

The only situations for the parallel model in which the processors P1 and P2 work or stay are:

(i) P1 works, P2 works

(ii) P1 stays, P2 works

(iii) P1 works, P2 stays

It is obvious that the situation (i) for which `action1(P1)` and `action2(P2)` will be called and executed in parallel, is described in an equivalent way in general left bidirectional parser by transitions: expand - shift, expand - reduce, reduce - shift, reduce - reduce. The situation (ii) corresponds to stay - shift and stay - reduce. The situation (iii) corresponds to expand - stay and reduce - stay. Finally, one of the processors will made the transitions of accepting and rejecting the input word. The nondeterministic behaviour of the general left bidirectional parser is realized in the parallel algorithm by introducing these backtrack points in `action1(P1)` and `action2(P2)`. If no backtrack step could be made, then the variable *exit* is true, so the input word is not accepted by the parser (and of course, it is not in the language of the input grammar).
So, the parallel Algorithm (PAR_LEFT) is correct. ■

We can say that it is possible to store into local memories of P1 and P2 the input word (and deleting it from the common memory). This does not

change the implementation because `w` is accessed in a read style. Now, the only variables which are in the common memory are `i1` and `i2`.

Another modification is to replace the variables `i1`, `i2` from the common memory, and make a direct link from P1 and P2 (and viceversa, of course). In that case, we can just send to the other processor the value of `i1`, respectively `i2`.

We point out a parallel algorithm which describes the **general right bidirectional parsing strategy.** Because it is similar to the general left bidirectional parsing strategy, we shall describe only the parts where this parallel algorithms differ.

The parallel algorithm (PAR_RIGHT) corresponding to the "right" model can now be given ($G = (V_N, V_T, S, P)$ is the input context free grammar):

```
begin
   read(n); read(w); i1:=1; i2:=n; push(Stack2,S);
   accept:=false; exit:=false;
   repeat in parallel
     action1(P1);
     if (i2>=i1) then action2(P2);
   until (i1>=i2) or (exit=true);
   if (i1>=i2) and (exit=false) then begin
     is_no_over:=false;
     while (is_no_over=false) and (exit=false) do
       if (Stack1=Stack2) then begin
         is_no_over:=true;
         accept:=true
       end
       else
         action1(P1)
   end;
   if (accept=true) then begin
     write('w is accepted and has the left syntactic analysis ');
     write(Output_tape2, Output_tape1);
   end
   else write('w is not accepted.');
end;
```

The procedures `action1(P1)` and `action2(P2)` are:

```
procedure action1(P1);
begin
   case
     if (∃ r= B → β ∈ P, β is in Stack1 starting from top) then begin
       /* reduce action */
       pop(Stack1,β);
       push(Stack1,B);
```

```
      push(Output_tape1, r);
    end
    if (i1<i2) then begin
      /* shift action */
      i1:=i1+1;
      push(Stack1,w[i1]);
    end
    otherwise: begin
      backtrack step is needed;
      if (all the backtrack steps are over) and
        (still no reduce or expand action could be made)
      then exit:=true
    end
end;

procedure action2(P2);
begin
  pop(Stack2,top);
  case top of
    (top∈ V_T) and (top=w[i2]):
      /* reduce action */
      if (i2>i1) then i2:=i2-1;
    top∈ V_N: begin
      /* expand action */
      find a production r : top → α ∈ P;
      if (there exists an r of this form) then begin
        push(Stack2,α);
        push(Output_tape2,r);
      end
    end;
    otherwise: begin
      backtrack step is needed;
      if (all the backtrack steps are over) and
        (still no reduce or expand action could be made)
      then exit:=true
    end
end;
```

Similar results related to the finiteness and correctness of Algorithm (PAR_RIGHT) can be shown in analogous way.

3 Deterministic subclasses of context free grammars

In this section we present some important subclasses of context free grammars for which there exist deterministic algorithms for solving the membership prob-

lem. The time complexity will be $\mathcal{O}(n)$, where n is the length of the input word.

Now, we give an important result which establishes the relation between a context free grammar G and its reverse $\widetilde{G}$.

Theorem 3.1 *Let $G = (V_N, V_T, S, P)$ be a context free grammar and the grammar $\widetilde{G} = (V_N, V_T, S, \widetilde{P})$ be its reverse. Then, for any $A \in V_N$, any derivation $\pi \in \{1, 2, ..., |P|\}^*$, we have:*

$$A \underset{G,lm}{\overset{\pi}{\Longrightarrow}} w\,\gamma \text{ iff } A \underset{\widetilde{G},rm}{\overset{\pi}{\Longrightarrow}} \widetilde{\gamma}\,\widetilde{w}$$

where $w \in V_T^*$, $\gamma \in V_N \cdot V^* \cup \{\lambda\}$.

Proof We shall proceed by induction on the number of derivations. The basis of induction ($|\pi| = 1$) can be obviously obtained. We shall show only the direct implication of that equivalence.
Inductive Step: Let $A \underset{G,lm}{\overset{\pi}{\Longrightarrow}} w\,\gamma$ be the left most derivation π, ($|\pi| \geq 1$), $w \in V_T^*$, $\gamma \in V_N \cdot V^* \cup \{\lambda\}$. Because $|\pi| \geq 1$, this means that there exists a grammar production r such that $\pi = \pi_1 r$. Now, we may rewrite the derivation:

$$A \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} w_1\,B\,\gamma_2 \underset{G,lm}{\overset{r}{\Longrightarrow}} w\,\gamma,\ r = B \rightarrow w_2\,\gamma_1,\ w_1\,w_2 = w,\ \gamma_1\,\gamma_2 = \gamma.$$

But $|\pi_1| < |\pi|$, so applying the inductive hypothesis, we obtain:

$$A \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} \widetilde{\gamma_2}\,w\,\widetilde{\gamma_1}$$

Now, because $\widetilde{w_1} \in V_T^*$, we continue with a right derivation applying in $\widetilde{G}$ the production $r = B \rightarrow \widetilde{\gamma_1}\,\widetilde{w_2}$:

$$A \underset{\widetilde{G},rm}{\overset{\pi_1}{\Longrightarrow}} \widetilde{\gamma_2}\,B\,\widetilde{w_1} \underset{\widetilde{G},rm}{\overset{r}{\Longrightarrow}} \widetilde{\gamma_2}\,\widetilde{\gamma_1}\,\widetilde{w_2}\,\widetilde{w_1} = \widetilde{\gamma}\,\widetilde{w}.$$

The other implication can similarly be obtained, so the proof of this theorem is complete. ■

3.1 $RR(k)$ grammars

For presenting $RR(k)$ grammars, we review the definition of $LL(k)$ grammars from [LeS68] (initially they were called $TD(k)$ grammars).

Definition 3.1 *We say that $G = (V_N, V_T, S, P)$ is a $LL(k)$ grammar (where $k \geq 0$) if for any two distinct derivations of the form:*

$$S \underset{lm}{\overset{*}{\Longrightarrow}} u\,A\,\alpha \underset{lm}{\Longrightarrow} u\,\beta_1\,\alpha \underset{lm}{\overset{*}{\Longrightarrow}} u\,v_1$$

$$S \underset{lm}{\overset{*}{\Longrightarrow}} u\,A\,\alpha \underset{lm}{\Longrightarrow} u\,\beta_2\,\alpha \underset{lm}{\overset{*}{\Longrightarrow}} u\,v_2$$

for which ${}^{(k)}v_1 = {}^{(k)}v_2$ then $\beta_1 = \beta_2$.

The name $RR(k)$ comes from: **R**ight to left scanning of the input constructing a **R**ightmost derivation in reverse, using **k** symbols of lookahead.

Definition 3.2 *Let G be a context free grammar and k be a natural number. We say that G is $RR(k)$ grammar if $\widetilde{G}$ is a $LL(k)$ grammar. A language L is $RR(k)$ if there exists a $RR(k)$ grammar which generates L.*

As a remark, if $G = (V_N, V_T, S, P)$ is a $RR(0)$ (reduced) grammar, then for any $A \in V_N$, there exists (exactly) at most one production of the form $A \to \alpha \in P$. Of course, if G is a reduced grammar and $RR(0)$, then its language is finite. That is why these grammars have no practical interest.

In [LeS68] is said that $RR(k)$ grammars may be obtained from $LL(k)$ "by reversing the roles of left and right one", but no precise definition has been given.

In order to define a parser for $RR(k)$ grammars, we shall give some definitions and results which are dual to $LL(k)$ grammars.

Theorem 3.2 *Any $RR(k)$ grammar is unambiguous.*

Proof Using Definition 3.2 and a result for $LL(k)$ grammars ([LeS68]). ∎

Theorem 3.3 *If G is a right recursive grammar, then there exists no natural number k such as G is a $RR(k)$ grammar.*

Proof Using Definitions 3.1 and 3.2. ∎

Theorem 3.4 *The $RR(k)$ languages form a strict infinite hierarchy:*

$$RR(0) \subset RR(1) \subset RR(2) \subset ... \subset RR(k) \subset RR(k+1) \subset ...$$

Proof Using Definitions 3.1 and 3.2. ∎

Lemma 3.1 *There exist context free languages which are not $RR(k)$ for any natural number k.*

Proof By showing that the language $L = \{a^n\, c\, b^n,\ a^{2n}\, d\, b^n \mid n \geq 1\}$ cannot be $RR(k)$, for any natural number k. ∎

In the following, we shall define a subclass of $RR(k)$ grammars (called $SRR(k)$) for which there exists an efficient characterization. Furthermore, the subclass $SRR(1)$ coincides with $RR(1)$. First, we shall describe some auxiliary sets of words useful for defining the $SRR(k)$ grammars.

Definition 3.3 *Let $G = (V_N, V_T, S, P)$ be a context free grammar, $\alpha \in V^+$, $A \in V_N$ and $k \in \mathbf{N}_+$. Then*

$$LAST_k(\alpha) = \{u \in V_T^* \mid |u| \leq k,\ \alpha \stackrel{*}{\Longrightarrow} v\, u,\ v \in V_T^*\};$$

$$PREVIOUS_k(A) = \{u \in V_T^* \mid S \stackrel{*}{\Longrightarrow} \alpha\, A\, \beta \text{ and } u \in LAST_k(\alpha)\}.$$

As a remark, we have to notice that for a given nonterminal symbol A, the set $PREVIOUS_k(A)$ represents the set of all terminal words of the length less than k which may occur in sentential forms, in front of the occurrence of A. We can immediately extend the sets $LAST_k$ to sets of words in this way:

$$\text{if } L \subseteq V^* \text{ then } LAST_k(L) = \bigcup_{\alpha \in L} LAST_k(\alpha).$$

Furthermore, if $\alpha \in V^+$ and $L \subseteq V^+$, we denote $\alpha L = \{\alpha x \mid x \in L\}$.

Definition 3.4 *A context free grammar $G = (V_N, V_T, S, P)$ is $SRR(k)$ if for any $A \in V_N$ and any two productions $A \to \beta_1$, $A \to \beta_2$ the following statement is fulfilled:*

$$LAST_k(PREVIOUS_k(A) \cdot \beta_1) \cap LAST_k(PREVIOUS_k(A) \cdot \beta_2) = \emptyset$$

The associated deterministic parser for this subclass of grammars can now be given. We have to mention that it has only one state, so we shall not consider the set of states as a component.

Definition 3.5 *Let $G = (V_N, V_T, S, P)$ be a $SRR(k)$ grammar, where $k \in \mathbf{N}$. Let $\mathcal{C} \subseteq \#V_T^* \times \#V^* \times \{1, 2, ..., |P|\}^* \cup \{ACC, REJ\}$ be the set of all possible configurations, where $\#$ is a special character (a new terminal symbol). The $PSRR_k(G)$* **parser** *is the pair $(\mathcal{C}_0, \vdash)$, where $\mathcal{C}_0 = \{(\#w, \#S, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**, *and $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ is the* **transition relation** *(sometimes denoted by $\underset{PSRR_k(G)}{\longmapsto}$) between configurations given by:*

1^0 *Expand:* $(\#u, \#\gamma A, \pi) \vdash (\#u, \#\gamma \beta, \pi r)$ *if* $r = A \to \beta \in P$ *and* $\#u^{(k)} \in LAST_k(\#PREVIOUS_k(A) \cdot \beta)$;

2^0 *Reduce:* $(\#u\, a, \#\gamma\, a, \pi) \vdash (\#u, \#\gamma, \pi)$

3^0 $(\#, \#, \pi) \vdash ACC$;

4^0 $(\#u, \#\gamma, \pi) \vdash REJ$ *if no transitions of type 1^0, 2^0 and 3^0 can be applied.*

Theorem 3.5 *Let G be a $SRR(k)$ grammar. Then the parser $PSRR_k(G)$ is deterministic, i.e. for any configuration at most one transition can be applied.*

Proof The expand transitions are deterministic according to Definition 3.4. The rest of the configurations are obviously distinct. ■

Theorem 3.6 *For any $k \in \mathbf{N}^+$ the class of $SRR(k)$ grammars is strictly included in the class of $SRR(k+1)$ grammars.*

Proof It is obvious why a $SRR(k)$ grammar is a $SRR(k+1)$ grammar. On the other hand, the grammar $G = (\{S\}, \{a\}, S, \{S \to a^k \mid a^{k+1}\})$ is a $SRR(k+1)$ grammar, but not a $SRR(k)$ grammar. ■

Theorem 3.7 *Any $SRR(k)$ grammar, where $k \in \mathbf{N}$, is unambiguous.*

Theorem 3.8 *A right recursive grammar is not a $SRR(k)$ grammar, for any $k \in \mathbf{N}^+$.*

More generally than the previous result, we can say that:

Theorem 3.9 *Let $G = (V_N, V_T, S, P)$ be a reduced context free grammar and $k \in \mathbf{N}^+$. Then G is a right recursive grammar iff there exists $w \in V_T^*$ for which the parser $PSRR_k(G)$ has an infinite number of configurations.*

3.1.1 $RR(1)$ grammars

$RR(1)$ grammars can be very useful in practice because the auxiliary sets $LAST_1$, $PREVIOUS_1$ (which from now on will be simply denoted by $LAST$ and $PREVIOUS$) may be computed in polynomial time complexity. According to Definition 3.3, we can rewrite $LAST$ and $PREVIOUS$ as follows:

If $\alpha \in V^+$ and $A \in V_N$ then:

$$LAST(\alpha) = \{a \mid a \in V_T,\ \alpha \stackrel{*}{\Longrightarrow} u\,a\} \cup \begin{cases} \{\lambda\} & \text{if } \alpha \stackrel{*}{\Longrightarrow} \lambda \\ \emptyset & \text{otherwise} \end{cases};$$

$$PREVIOUS(A) = \{a \mid a \in V_T \cup \{\lambda\},\ S \stackrel{*}{\Longrightarrow} \alpha\, A\, \beta,\ a \in LAST(\alpha)\}.$$

The next result establishes the equivalence between $SRR(1)$ and $RR(1)$ grammars.

Theorem 3.10 *A context free grammar $G = (V_N, V_T, S, P)$ is $RR(1)$ iff for any $A \in V_N$ and any two distinct productions $A \to \beta_1 \mid \beta_2$ the following statements are fulfilled:*

$$LAST(PREVIOUS(A) \cdot \beta_1) \cap LAST(PREVIOUS(A) \cdot \beta_2) = \emptyset.$$

Proof According to Definition 3.2 and a similar result for $LL(1)$ grammars ([LeS68]). ■

For computing the sets $LAST$ and $PREVIOUS$, we need to define some binary relations over $V \times V$.

Definition 3.6 *Let G be a context free grammar. We define the following binary relations over $V \times V$:*

1. X `begin` A *if* $\exists\ A \to \alpha\, X\, \beta \in P$ *and* $\alpha \stackrel{*}{\Longrightarrow} \lambda$;
2. X `end` A *if* $\exists\ A \to \alpha\, X\, \beta \in P$ *and* $\beta \stackrel{*}{\Longrightarrow} \lambda$;
3. X `followed_by` Y *if* $\exists\ A \to \alpha\, X\, \beta\, Y\, \gamma \in P$ *and* $\beta \stackrel{*}{\Longrightarrow} \lambda$;
4. X `terminal` Y *if* $X,\ Y \in V_T$ *and* $X = Y$;

5. X nonterminal Y *if* $X,\ Y \in V_N$ *and* $X = Y$;

6. $\texttt{last} = \texttt{terminal} \circ \texttt{end}^* \circ \texttt{nonterminal}$;

7. $\texttt{previous} = \texttt{terminal} \circ \texttt{end}^* \circ \texttt{followed_by} \circ (\texttt{begin}^*)^{-1} \circ \texttt{nonterminal}$.

Theorem 3.11 *Let* $G = (V_N, V_T, S, P)$ *be a reduced context free grammar and* $a \in V_T$, $X \in V_N$. *Then the following statements are fulfilled:*

(i) $a \in LAST(X)$ *iff* $a\,\texttt{last}\,X$; $\lambda \in LAST(X)$ *iff* $X \stackrel{*}{\Longrightarrow} \lambda$;

(ii) $a \in PREVIOUS(X)$ *iff* $a\,\texttt{previous}\,X$; $\lambda \in PREVIOUS(X)$ *iff* $X\,\texttt{begin}^*\,S$.

Example 3.1 *Let us consider the grammar given by the following productions:*
1. $S \to E$ 2. $S \to B$ 3. $E \to \lambda$ 4. $B \to a$
5. $B \to b\,C\,S\,e$ 6. $C \to \lambda$ 7. $C \to C\,S\,;$

First, we compute the null nonterminal symbols, i.e. $X \in V_N$ *is a null symbol if* $\exists\ X \stackrel{*}{\Longrightarrow} \lambda$. *For our grammar the set of null symbols is* $\{S, E, C\}$. *We compute the auxiliary binary relations:*

$\texttt{end} = \{(E,S),(B,S),(a,B),(e,B),(;,C)\}$;
$\texttt{end}^* = id_V \cup \texttt{end} \cup \{(a,S),(e,S)\}$;
$\texttt{last} = \{(a,B),(e,B),(;,C),(a,S),(e,S)\}$;

So, the sets LAST for the nonterminals symbols are:

X	S	E	B	C
$LAST$	$\{a,e,\lambda\}$	$\{\lambda\}$	$\{a,e\}$	$\{;,\lambda\}$

We continue with the other relations:

$\texttt{begin}^{-1} = \{(S,E),(S,B),(B,a),(B,b),(C,C),(C,S),(C,;)\}$;
$(\texttt{begin}^{-1})^* \circ \texttt{nonterminal} = id_{V_N} \cup \{(S,E),(S,B),(C,S),(C,E),(C,B)\}$;
$\texttt{followed_by} = \{(b,C),(b,S),(b,e),(C,S),(C,e),(S,e),(C,;),(S,;)\}$;
$\texttt{previous} = \{(b,C),(b,S),(b,E),(b,B),(;,S),(;,E),(;,B)\}$.

So, the sets PREVIOUS for the nonterminal symbols are:

X	S	E	B	C
$PREVIOUS$	$\{b,;,\lambda\}$	$\{b,;,\lambda\}$	$\{b,;\}$	$\{b,\lambda\}$

According to Theorem 3.10, we check if our grammar is $RR(1)$:

$LAST(PREVIOUS(S)\cdot E) = \{b,;,,\lambda\}$;
$LAST(PREVIOUS(S)\cdot B) = \{a,e\}$;
$LAST(PREVIOUS(B)\cdot a) = \{a\}$;
$LAST(PREVIOUS(B)\cdot b\,C\,S\,e) = \{e\}$;
$LAST(PREVIOUS(C)\cdot \lambda) = \{b,\lambda\}$;
$LAST(PREVIOUS(C)\cdot C\,S\,;) = \{;\}$;

Therefore, G *is a* $RR(1)$ *grammar.*

Next, we present the parser attached to a $RR(1)$ grammar.

Definition 3.7 *Let $G = (V_N, V_T, S, P)$ be a $RR(1)$ grammar and $LAST$, $PREVIOUS$ the corresponding sets of words (defined above). We denote by $\mathcal{C} \subseteq \#V_T^* \times \#V^* \times \{1,2,...,|P|\}^* \cup \{ACC, REJ\}$ the set of all possible configurations, where $\#$ is a special character (a new terminal symbol). The $PRR_1(G)$* **parser** *is the pair $(\mathcal{C}_0, \vdash)$, where $\mathcal{C}_0 = \{(\#w, \#S, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**, *and $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ is the* **transition relation** *(sometimes denoted by $\underset{PRR_1(G)}{\longmapsto}$) between configurations given by:*

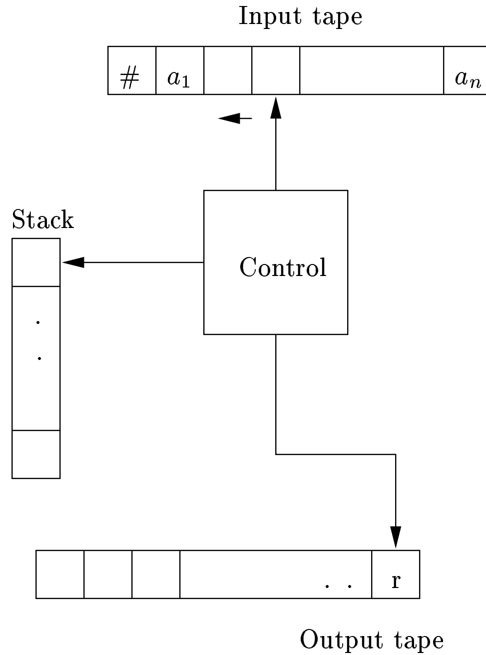
1^0 *Expand:* $(\#u, \#\gamma\, A, \pi) \vdash (\#u, \#\gamma\, \beta, \pi\, r)$ *if* $r = A \to \beta \in P$ *and* $\#u^{(1)} \in LAST(\#PREVIOUS(A) \cdot \beta)$;

2^0 *Reduce:* $(\#u\, a, \#\gamma\, a, \pi) \vdash (\#u, \#\gamma, \pi)$

3^0 $(\#, \#, \pi) \vdash ACC$;

4^0 $(\#u, \#\gamma, \pi) \vdash REJ$ *if no transitions of type 1^0, 2^0 and 3^0 can be applied.*

The parser $PRR_1(G)$ is similar to a $LL(1)$ parser, but the difference is that it scans the input word from the end to its start. It can push or pop strings in the stack. The output tape will contain the right syntactic analysis. It returns "ACC" or "REJ" depending on whether the input word is accepted or not. Again the parser has only one state.



In the following, we shall present a parsing algorithm for $RR(1)$ grammars.

The Algorithm (Parsing-$RR(1)$)

Input: The $RR(1)$ grammar $G = (V_N, V_T, S, P)$, the sets $LAST, PREVIOUS$ corresponding to the nonterminal symbols and the word $w \in V_T^*$;
Output: The right syntactic analysis if $w \in L(G)$; otherwise the message: 'w is not accepted'.
Method: It is assumed that we can use the following predefined procedures:

- `pop(stack,top)` - the value of `top` will be the value of the first symbol from `stack`;
- `push(stack,X)` - push in the top of `stack` the value of `X`;
- `push(Output_tape,r)` - push in the top of `Output_tape` the value of `r`.

The main program is:

```
begin
   read(w); push(stack,"#S"); i:= |w|+1;
   accept:=false; is_over:=false;
   put in the input tape the string "#w";
   repeat
     pop(stack,top);
     remove the top of the stack;
     if (top∈ V_N) then begin
       /* expand action */
       find r=top→ α ∈ P such that w[i]∈ LAST(PREVIOUS(top)α);
       if (does not exist such a production) then
         is_over:=true; /* reject action */
       else begin
         push(stack,α);
         push(Output_tape,r);
       end
     end
     else if (top=w[i] and i>1) then
         /* reduce action */
         i:=i-1;
       else begin
         is_over:=true;
         if (top=#) and (i=1) then
           /* accept action */
           accept:=true
       end;
   until (is_over=true);
   if (accept=true) then
     write('w is accepted and has the right syntactic analysis ',Output_tape)
   else write('w is not accepted.')
end.
```

Theorem 3.12 *(correctness and complexity of Algorithm (Parsing-$RR(1)$))*
Algorithm (Parsing-$RR(1)$) is correct and has the time complexity of $\mathcal{O}(|w|)$, where w is the input word.

Proof The correctness follows directly from Definition 3.2 and [LeS68]. The number of iterations of the cycle "`repeat... until`" is $\mathcal{O}(m \cdot |w|)$, where m is a constant depending of G. During an iteration, Algorithm (Parsing-$RR(1)$) makes an expand or a reduce action. The number of consecutively expand actions is finite because the input grammar is finite. A reduce action means reading a letter from w. Therefore the time complexity of Algorithm (Parsing-$RL(0)$) is $\mathcal{O}(|w|)$. ■

Example 3.2 *We reconsider the grammar from Example 3.1. Let $w = b\,a\,;\,e$ be the input word for Algorithm (Parsing-$RR(1)$). We obtain:*
$(\#b\,a\,;\,e, \#S, \lambda) \overset{1^0}{\vdash} (\#b\,a\,;\,e, \#B, [2]) \overset{1^0}{\vdash} (\#b\,a\,;\,e, \#b\,C\,S\,e, [2,5]) \overset{2^0}{\vdash}$
$(\#b\,a\,;, \#b\,C\,S, [2,5]) \overset{1^0}{\vdash} (\#b\,a\,;, \#b\,C\,E, [2,5,1]) \overset{1^0}{\vdash} (\#b\,a\,;, \#b\,C, [2,5,1,3]) \overset{1^0}{\vdash}$
$(\#b\,a\,;, \#b\,C\,S\,;, [2,5,1,3,7]) \overset{2^0}{\vdash} (\#b\,a, \#b\,C\,S, [2,5,1,3,7]) \overset{1^0}{\vdash}$
$(\#b\,a, \#b\,C\,B, [2,5,1,3,7,2]) \overset{1^0}{\vdash} (\#b\,a, \#b\,C\,a, [2,5,1,3,7,2,4]) \overset{2^0}{\vdash}$
$(\#b, \#b\,C, [2,5,1,3,7,2,4]) \overset{1^0}{\vdash} (\#b, \#b, [2,5,1,3,7,2,4,6]) \overset{2^0}{\vdash}$
$(\#b, \#b, [2,5,1,3,7,2,4,6]) \overset{3^0}{\vdash} ACC$. *Therefore, $w \in L(G)$ and the right syntactic analysis for it is $\pi = [2,5,1,3,7,2,4,6]$. The notation for the terminal symbols of this grammar comes from programming languages:*
b - **begin**, e - **end**, a - **statement** *and* ; - **end of statement**.

3.2 $RL(k)$ grammars

For presenting $RL(k)$ grammars, we review the definition of $LR(k)$ grammars ([Knu65]).

Definition 3.8 *We say that $G = (V_N, V_T, S, P)$ is a $LR(k)$ grammar (where $k \geq 0$) if for any two distinct derivations of the form:*

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha\, A\, u \underset{rm}{\Longrightarrow} \alpha\, \beta\, u$$

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha'\, A'\, u' \underset{rm}{\Longrightarrow} \alpha'\, \beta'\, u' = \alpha\, \beta\, v$$

for which ${}^{(k)}u = {}^{(k)}v$ then $\alpha = \alpha'$, $A = A'$ and $\beta = \beta'$.

The name $RL(k)$ comes from: **R**ight to left scanning of the input constructing a **L**eftmost derivation in reverse, using **k** symbols of lookahead.

Definition 3.9 *Let G be a context free grammar and k be a natural number. We say that G is a $RL(k)$ grammar if $\widetilde{G}$ is a $LR(k)$ grammar. A language L is $RL(k)$ if there exists a $RL(k)$ grammar which generates L.*

At the end of the paper [Knu65] there has been presented an example of a $RL(0)$ grammar, but the only "definition" for $RL(0)$ grammars was that these are obtained from $LR(0)$ by a "asymmetric property". Later, in [Knu71], the author has referred again to $RL(k)$ and $RR(k)$ grammars in an informal way, too.

In order to define a parser for $RL(k)$ grammars, we shall again need some definitions and results which are dual to $LR(k)$ grammars.

Theorem 3.13 *Any $RL(k)$ grammar is unambiguous.*

Proof Using Definition 3.9 and a similar result for $LR(k)$ grammars. ■

Theorem 3.14 *([Knu65]) There exist $RL(0)$ languages which cannot be $LR(k)$ languages.*

Proof Let G be the grammar:

$$S \to A\,c \,|\, B \quad A \to a\,A\,b\,b \,|\, a\,b\,b \quad B \to a\,B\,b \,|\, a\,b$$

Obviously, G is a $RL(0)$ grammar because $\widetilde{G}$ is a $LR(0)$ grammar. We observe that $L(G) = \{a^n\, b^{2n}\, c,\ a^n\, b^n \mid n \geq 1\}$ is not a deterministic context free language. But ([Knu65]) a context free language can be generated by a $LR(k)$ grammar if and only if it is deterministic. So, we cannot find an equivalent $LR(k)$ grammar to G. ■

3.2.1 $RL(0)$ grammars

Definition 3.10 *Let $G = (V_N, V_T, S, P)$ be a context free grammar and $\bullet \notin V$ a new symbol (called* **point***). An $RL(0)$* **item** *for G is a construction $A \to \beta_1 \bullet \beta_2$, where $A \to \beta_1\beta_2 \in P$. The set of all items of G is denoted by $I(G)$.*

Example 3.3 *For the production $S \to x_1\, x_2 \dots x_n$ there exist $n+1$ items, such as:*

$$S \to x_1\, x_2 \dots x_n \bullet,\ \ S \to x_1\, x_2 \dots x_{n-1} \bullet x_n,\ \ \dots\ S \to \bullet x_1\, x_2 \dots x_n$$

Definition 3.11 *An $RL(0)$ item by the form $A \to \bullet\beta$ is called* **complete***, i.e. the point is at the beginning of the production.*

Definition 3.12 *A* **viable suffix** *for G is a word $\gamma \in V^*$ for which there exists a derivation $S \overset{*}{\underset{lm}{\Longrightarrow}} u\, A\, \alpha \underset{lm}{\Longrightarrow} u\, \beta\alpha$, and γ is a suffix for $\beta\, \alpha$ (i.e. there exists $\gamma' \in V^*$ such that $\beta\, \alpha = \gamma'\, \gamma$).*

Example 3.4 *Let us consider the left derivation:*

$$A \Longrightarrow a\, A\, B \Longrightarrow a\, c\, d\, B$$

According to Definition 3.12, the viable suffixes for $a\,c\,d\,B$ are λ, B, $d\,B$, $c\,d\,B$ and $a\,c\,d\,B$.

Definition 3.13 *An item $A \to \beta_1 \centerdot \beta_2$ is* **valid** *for the viable suffix γ if there exists a derivation:*

$$S \underset{lm}{\overset{*}{\Longrightarrow}} u\, A\, \alpha \underset{lm}{\Longrightarrow} u\, \beta_1\, \beta_2\, \alpha \text{ and } \gamma = \beta_2\, \alpha$$

The set of all valid items for the viable suffix γ is denoted by $I(\gamma)$.

Example 3.5 *Let us consider the grammar:*

$A \to a\, A\, B \mid c\, d \quad B \to b\, e \mid f$

For the suffix "λ", there are several valid items, such as:

$$A \to a\, A\, B \centerdot,\ A \to c\, d \centerdot,\ B \to b\, e \centerdot,\ B \to f \centerdot$$

For the suffix "d", there is only one valid item: $A \to c \centerdot d$.
For the suffix "$b\, e$", there is only one valid item, too: $B \to \centerdot b\, e$.

Theorem 3.15 *(characterization of $RL(0)$ grammars)*
A reduced grammar $G = (V_N, V_T, S, P)$ (and in which S does not appear in the right side of the productions) is $RL(0)$ if and only if for all viable suffixes γ, the set of all valid items for γ (denoted by $I(\gamma)$) satisfies the conditions:

(i) $I(\gamma)$ contains no two (or more than two) distinct complete items;

(ii) if $A \to \centerdot \alpha \in I(\gamma)$ then $I(\gamma)$ contains no item of the form
$B \to \beta_1\, a \centerdot \beta_2,\ a \in V_T$.

Proof Similar to the corresponding proof for $LR(0)$ grammars ([Knu65]). ∎

Theorem 3.16 *Let G be a reduced context free grammar. The set of all viable suffixes of the grammar G (generally an infinite set) is a regular language.*

Proof Similar to the corresponding proof for $LR(0)$ grammars [Knu65]. We present only the idea. Starting from the reduced context free grammar $G = (V_N, V_T, S, P)$, we shall construct a nondeterministic finite automaton with $\lambda-$transitions which accepts the set of all viable suffixes. Let $M = (I, V, \delta, q_0, I)$ be an automaton for which:

- $I = \{q_0\} \cup \{A \to \beta_1 \centerdot \beta_2 \mid A \to \beta_1\beta_2 \in P\}$
- δ is defined as follows:
  (i) $\delta(A \to \beta_1\, B \centerdot \beta_2, \lambda) = \{B \to \beta \centerdot \mid B \to \beta \in P\}$;
  (ii) $\delta(A \to \beta_1\, X \centerdot \beta_2, X) = \{A \to \beta_1 \centerdot X\, \beta_2\}$;
  (iii) $\delta(A \to \alpha\, a \centerdot \beta, \lambda) = \delta(A \to \alpha\, X \centerdot \beta, Y) = \emptyset,\ Y \in V_N,\ X \neq Y,\ a \in V_T$.

We put q_0 as a notation for the initial state, which is in fact the item $S' \to S \centerdot$. Here S' is a new symbol and $S' \to S$ is a new production. Sometimes, if S does not occur on the right side of the grammar productions, we may choose $S' = S$ (in that case, the "initial state" will be a set of items of the form $S \to \alpha \centerdot$, where $S \to \alpha \in P$). ∎

Example 3.6 *Let us consider the context free grammar G given by the productions:*

$$S \to a\,A\,d \,|\, b\,A\,B,\;\; A \to c\,A \,|\, c,\;\; B \to b$$

We shall construct the equivalent automaton M with $\lambda-$transitions. Then we check if G is a $RL(0)$ grammar using Theorem 3.15. The two conditions from Theorem 3.15 may be translated in the automaton graph into:

(i) the automaton graph contains no two vertices $A \to \,\centerdot\alpha$, $B \to \,\centerdot\beta$ $(A \to \alpha \neq B \to \beta)$, such that the paths from q_0 to these vertices are labelled with the same word;

(ii) the automaton graph contains no two vertices $A \to \,\centerdot\alpha$, $B \to \beta_1\, a \centerdot\beta_2$, $a \in V_T$ such that the paths from q_0 to these vertices are labelled with the same word.

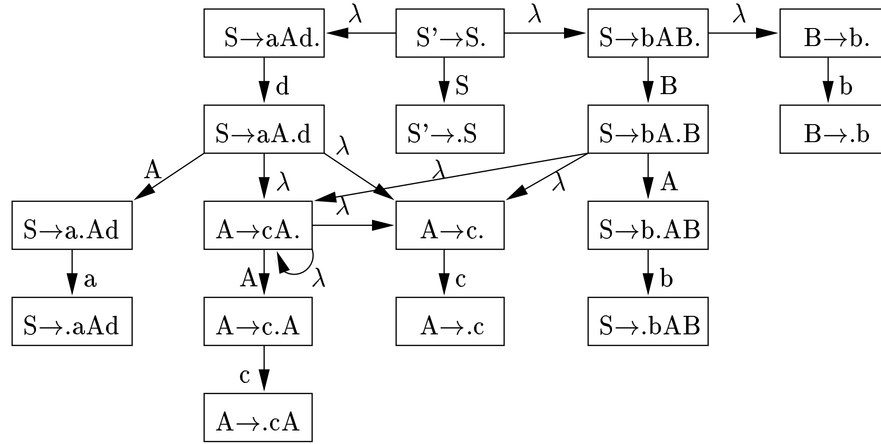


Figure 4. The automaton with λ-transitions of viable suffixes

We can observe that the graph of the automaton M satisfies the previous conditions (i) and (ii), so G is a $RL(0)$ grammar.

However, the conditions presented in Example 3.6 are hard to check on this automaton with $\lambda-$transitions. We may attach an equivalent deterministic automaton $M' = (T, V, \delta', t_0, T)$, where $T \subseteq 2^I$ (one state for the deterministic automaton is a subset of items), t_0 contains all items accessible from $q_0 = S' \to S\,\centerdot$ using $\lambda-$transitions, and $\delta' : T \times V \to T$ is partially defined.

But M accepts the set of all viable suffixes and $A \to \beta_1 \centerdot \beta_2 \in \delta(q_o, \gamma)$ if and only if $A \to \beta_1 \centerdot \beta_2$ is valid for γ. This implies (in M') that a state $t \in T$ contains exactly all valid items for a certain viable suffix. Therefore, the two conditions above (for deciding if G is a $RL(0)$ grammar) can be rewritten as:

(i) any state $t \in T$ contains at most a complete item;

(ii) for any state $t \in T$ which contains a complete item, t must contain no item in which a terminal symbol is followed by $\scriptstyle\blacksquare$.

These two conditions are easy to check for M'. In the following, we shall indicate the way of constructing M' (called the $RL(0)$ **automaton** of grammar G). First, we shall describe a function, called *closure*, which has as input a set of items t and returns in the output all states (set of items) accessible from t with $\lambda-$paths.

The Algorithm (Closure)

Input: $G = (V_N, V_T, S, P)$ a context free grammar and $t \subseteq I$ a set of items;
Output: $closure(t) = \{t'' \in I \mid \delta(t, \lambda) = t''\}$ (T is a notation for a subset of I);
Method:

```
function closure(t):T;
begin
  t':=t; flag:=true;
  while (flag=true) do begin
    flag:=false;
    for (all A → α B▪β ∈t') do
      for (all B → γ ∈ P) do
        if (B → γ▪ ∉t') then begin
          t':=t'∪{B → γ▪};
          flag:=true
        end
  end;
  return(t');
end;
```

Lemma 3.2 *Let $M = (I, V, \delta, q_0, I)$ be the automaton with $\lambda-$transitions associated to grammar G and let $t \subseteq I$ be a set of items. Then, using Algorithm (Closure) we obtain as output $closure(t) = \{t'' \in I \mid \delta(t, \lambda) = t''\}$.*

Now, we are ready to indicate an algorithm for constructing the $RL(0)$ automaton for grammar G.

The Algorithm ($RL(0)$-Automaton)

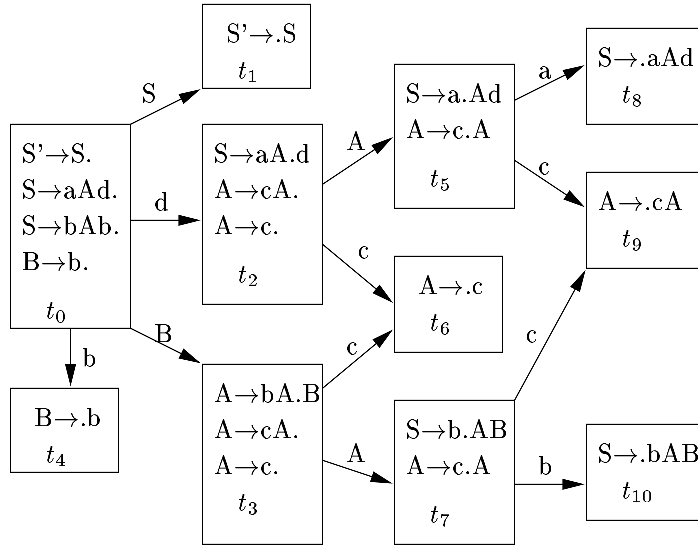
Input: $G = (V_N, V_T, S, P)$ a context free grammar (augmented with the production $S' \to S$);
Output: $M' = (T, V, \delta', t_0, T)$ a deterministic automaton equivalent to M;
Method:

```
begin
  t0:=closure(S' → S▪); T:= {t0}; marcat[t0]:=false;
  while (∃t ∈ T and not(marcat[t])) do begin
    for (all X ∈ V) do begin
```

```
        t':=∅;
        for (all A → αX.β ∈ t) do t':=t' ∪ {A → α.Xβ};
        if (t' ≠ ∅) then begin
          t':=closure(t');
          if (t' ∉ T) then begin
            T:=T ∪ {t'};
            marcat[t']:=false;
          end;
          δ'(t, X):=t';
        end
      end;
      marcat[t]:=true;
    end
  end;
```

Lemma 3.3 *The Algorithm (RL(0)-Automaton) is correct, i.e. M' is a deterministic automaton equivalent to M.*

Example 3.7 *We consider the same grammar as in Example 3.6. Using Algorithm (RL(0)-Automaton) we shall construct the deterministic automaton M' (we shall number the states of the automaton M' in a breadth first search manner):*

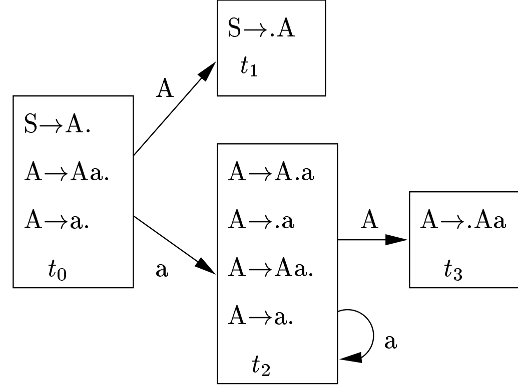


Now, checking the associated conditions for M', we can say that G is a $RL(0)$ grammar.

Example 3.8 *Let us consider the grammar*

$$G = (\{S, A\}, \{a\}, S, \{S \to A,\ A \to A\,a \,|\, a\}).$$

The deterministic automaton M' corresponding to G is:



We can see that the state t_2 does not satisfy the conditions for $RL(0)$ grammars because it contains the complete item $A \to \centerdot a$ and the item $A \to Aa \centerdot$ in which the terminal a is followed by $\centerdot$.

We can adapt the $LR(0)$ language characterization theorem to characterize $RL(0)$ languages (Theorem 13.3.1, [Har78]).

Theorem 3.17 *($RL(0)$ language characterization theorem)*
Let $L \subseteq \Sigma^*$, where Σ is an arbitrary alphabet. The following four statements are equivalent:

a) L is an $RL(0)$ language;

b) $\widetilde{L} \subseteq \Sigma^*$ is a deterministic context free language and for all $x \in \Sigma^+$, $w,\ y \in \Sigma^*$, if $w \in \widetilde{L}$, $wx \in \widetilde{L}$, and $y \in \widetilde{L}$, then $yx \in \widetilde{L}$;

c) there exists a deterministic pushdown automaton $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where $F = \{q_f\}$ and there exists $Z_f \in \Gamma$ such that

$$\widetilde{L} = T(A, Z_f) = T(A, \Gamma) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \overset{*}{\vdash} (q_f, \lambda, Z_f)\};$$

d) there exist strict deterministic languages L_0 and L_1 such that $\widetilde{L} = L_0 L_1^*$.

Next, we present the parser attached to a $RL(0)$ grammar.

Definition 3.14 *Let $G = (V_N, V_T, S, P)$ be a $RL(0)$ grammar and $M' = (T, V, \delta', t_0, T)$ the corresponding deterministic automaton. We denote by $\mathcal{C} \subseteq \#V_T^* \times (T \cdot V_T)^* t_0 \times \{1, 2, ..., |P|\}^* \cup \{ACC, REJ\}$ the set of all possible configurations, where $\#$ is a special character (a new terminal symbol). The $RL(0)$* **parser** *(denoted by $PRL_0(G)$) is the pair $(\mathcal{C}_0, \vdash)$, where the set $\mathcal{C}_0 = \{(\#w, t_0, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**, *and $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ is the* **transition relation** *(sometimes denoted by $\underset{PRL_0(G)}{\longmapsto}$) between configurations given by:*

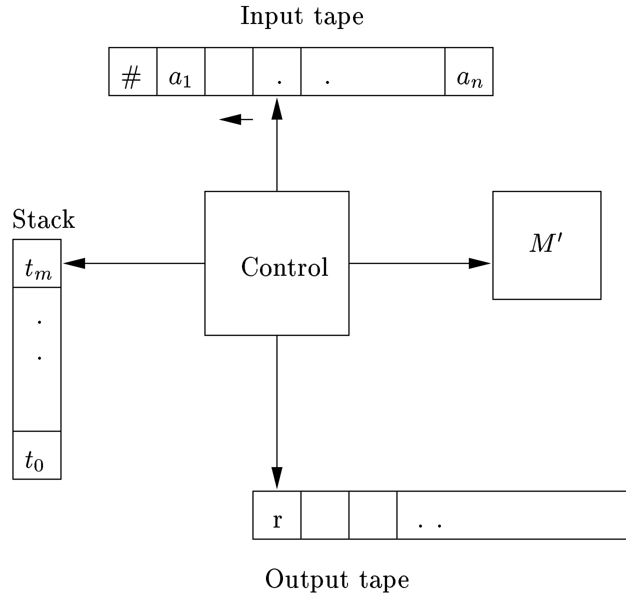
1^0 *Shift:* $(\#u\,a, t\,\sigma, \pi) \vdash (\#u, t'\,a\,t\,\sigma, \pi)$ *if* $\delta'(t,a) = t'$;

2^0 *Reduce:* $(\#u, t'\,\sigma\,t\,\sigma, \pi) \vdash (\#u, t''\,A\,t\,\sigma, r\,\pi)$ *if* $A \to {\scriptstyle\blacksquare}\beta \in t'$, $|\sigma'\ t'| = |\beta|$, $r = A \to \beta$, $t'' = \delta'(t, A)$;

3^0 *Accept:* $(\#, \sigma, \pi) \vdash ACC$ *if* $S' \to {\scriptstyle\blacksquare}A \in t_1$*;*

4^0 *Reject:* $(\#u, \sigma, \pi) \vdash REJ$ *if no transitions of type* 1^0, 2^0 *and* 3^0 *can be applied.*

The parser $PRL_0(G)$ is similar to the $LR(0)$ parser, but the difference is that it scans the input word from the end to its start. It can push or pop strings in the stack using δ' according to automaton M'. The output tape will contain the left syntactic analysis. It returns "ACC" or "REJ" depending on whether the input word is accepted or not.



In the following, we shall present the parsing algorithm for $RL(0)$ grammars.

The Algorithm (Parsing-$RL(0)$)

Input: The $RL(0)$ grammar $G = (V_N, V_T, S, P)$, the corresponding automaton $M' = (T, V, \delta', t_0, T)$, and the word $w \in V_T^*$;
Output: The left syntactic analysis if $w \in L(G)$; otherwise the message: 'w is not accepted'.
Method: It is assumed that we can use the following predefined procedures:

- `pop(stack,top)` - the value of `top` will be the value of the first symbol from `stack` (without removing the top of the `stack`);

- `push(stack,X)` - push in the top of `stack` the value of `X`;
- `push(Output_tape,r)` - push in the top of `Output_tape` the value of `r`.

The main program is:

```
begin
   read(w);push(stack,t_0);i:=|w |+1;
   accept:=false;is_over:=false;
   put in the input tape the string "#w";
   repeat
     pop(stack,t);
     if (δ'(t,w[i])≠ ∅ then begin
       /* shift action */
       push(stack,w[i]);
       push(stack,δ'(t,w[i]));
       i:=i-1;
     end
     else
       if (A → .X_1 X_2 ... X_m ∈t) then begin
         if (A = S') then begin
           /* accept action */
           push(Output_tape,S' → X_1 X_2 ... X_n);
           is_over:=true;
           accept:=true
         end;
         else begin
           /* reduce action */
           remove the first 2 * m symbols from the stack;
           pop(stack,t');
           t":=δ'(t',A);
           push(stack,A);
           push(stack,t");
           push(Output_tape,A → X_1 X_2 ... X_n)
         end
       end
       else is_over:=true /* reject action */
   until (is_over=true);
   if (accept=true) then
     write('w is accepted and has the left syntactic analysis ', Output_tape)
   else write('w is not accepted.')
end.
```

Theorem 3.18 *(correctness and complexity of Algorithm (Parsing-RL(0)))*
Algorithm (Parsing-RL(0)) is correct and has the time complexity of $\mathcal{O}(|w|)$, where w is the input word.

Proof The correctness follows directly from Definition 3.9 and [Knu65]. The number of iterations of the cycle "`repeat`... `until`" is $\mathcal{O}(m \cdot |w|)$, where m is a constant depending of G. During an iteration, Algorithm (Parsing-$RL(0)$) makes a shift or a reduce action. The number of consecutively reduce actions is finite because the input grammar is finite. A shift action means reading a letter from w. So the time complexity of Algorithm (Parsing-$RL(0)$) is $\mathcal{O}(|w|)$. ■

Example 3.9 *We reconsider the grammar from Examples 3.6 and 3.7. Let $w = b\,c\,c\,b$ be the input word for Algorithm (Parsing-RL(0)). We obtain:*

$$(\#b\,c\,c\,b, t_0, \lambda) \overset{1^0}{\vdash} (\#b\,c\,c, t_4\, b\, t_0, \lambda) \overset{2^0}{\vdash} (\#b\,c\,c, t_3\, B\, t_0, [5]) \overset{1^0}{\vdash} (\#b\,c, t_6\, c\, t_3\, B\, t_0, [5]) \overset{2^0}{\vdash}$$
$$(\#b\,c, t_7\, A\, t_3\, B\, t_0, [4,5]) \overset{1^0}{\vdash} (\#b, t_9\, c\, t_7\, A\, t_3\, B\, t_0, [4,5]) \overset{2^0}{\vdash} (\#b, t_7\, A\, t_3\, B\, t_0, [3,4,5])$$
$$\overset{1^0}{\vdash} (\#, t_{10}\, b\, t_7\, A\, t_3\, B\, t_0, [3,4,5]) \overset{2^0}{\vdash} (\#, t_1\, S\, t_0, [2,3,4,5]) \overset{3^0}{\vdash} ACC$$

Therefore, $w \in L(G)$ and the left syntactic analysis for it is $\pi = [2,3,4,5]$.

3.2.2 $SRL(1)$ grammars

Sometimes, for a given context free grammar G which is not an $RL(0)$ grammar, the *conflicts* reduce - reduce, reduce - shift from the $RL(0)$ automaton can be solved. In this sense, we shall define *simple* $RL(1)$ grammars (denoted by $SLR(1)$).

Definition 3.15 *The context free grammar $G = (V_N, V_T, S, P)$ is $SRL(1)$ if for any state $t \in T$ from the $RL(0)$ automaton $M' = (T, V, \delta, t_0, T)$, the following statements are fulfilled:*

(i) if $A \to \centerdot\alpha \in t$ and $B \to \centerdot\beta \in t$ then
$PREVIOUS(A) \cap PREVIOUS(B) = \emptyset$;

(ii) if $A \to \centerdot\alpha \in t$ and $B \to \beta\, a \centerdot \gamma \in t$ then $a \notin PREVIOUS(A)$.

Next, we shall define a relation $ACTION : T \to V_T \cup \{\#\}$ (sometimes called $SRL(1)$-table), which has the goal to decide the removing of the conflicts reduce - reduce, reduce - shift. This relation is constructed using the $RL(0)$ automaton and the sets $PREVIOUS$ by the following algorithm:

The Algorithm $SRL(1)$-table:

Input: The context free grammar $G = (V_N, V_T, S, P)$, the $RL(0)$ automaton $M' = (T, V, \delta, t_0, T)$, the sets $PREVIOUS(A)$, $\forall\ A \in V_N$.
Output: The relation $ACTION(t, a)$, $\forall\ t \in T$, $\forall\ a \in V_T \cup \{\#\}$.
Method:

```
begin
  for (all t ∈ T) do begin
    for (all A → •α ∈ t and A ≠ S') do begin
      for (a ∈ PREVIOUS(A)) do
        ACTION(t, a):=reduce_r, where r = A → α ∈ P;
```

```
        if (λ ∈ PREVIOUS(A)) then
          ACTION(t,#):=reduce_r, where r = A → α ∈ P
      end;
      for (all B → β a.γ ∈ t, a ∈ V_T) do
        ACTION(t,a):=shift t_k, where t_k = δ(t,a);
      if (S' → .S ∈ t) then
        ACTION(t,#):=ACC_r, where r = S' → S ∈ P
    end
  end.
```

It is obvious to remark that if the relation $ACTION$ is well defined, i.e. $|ACTION(t,a)| \leq 1$, $\forall\, t \in T$, $\forall\, a \in V_T$, then the two conditions from Definition 3.15 are fulfilled (so G is a $SRL(1)$ grammar), and viceversa.

In the following, we present the $SRL(1)$ parsing algorithm. Because it is similar to (Parsing-$RL(0)$) Algorithm, we present only the main program:

The Algorithm (Parsing-$SRL(1)$)

```
begin
  read(w);push(stack,t_0);i:=|w |+1;
  accept:=false;is_over:=false;
  put in the input tape the string "#w";
  repeat
    pop(stack,t);
    if (ACTION(t,w[i])= ACC_r) then begin
      /* accept action */
        push(Output_tape,r);
        is_over:=true;
        accept:=true
    end
    else
      if (ACTION(t,w[i])= shift t_k then begin
        /* shift action */
        push(stack,t_k);
        i:=i-1;
      end
      else
        if (ACTION(t,w[i])= reduce_r then begin
          /* reduce action */
          let A → α be the production r;
          remove the first |α| symbols from the stack;
          pop(stack,t');
          push(stack,δ(t',A));
          push(Output_tape,r)
        end
        else is_over:=true /* reject action */
```

```
   until (is_over=true);
   if (accept=true) then
     write('w is accepted and has the left syntactic analysis ', Output_tape)
   else write('w is not accepted.')
end.
```

Theorem 3.19 *(correctness and complexity of Algorithm (Parsing-$SRL(1)$))*
Algorithm (Parsing-$SRL(1)$) is correct and has the time complexity $\mathcal{O}(|w|)$, where w is the input word.

Proof Using Definition 3.15, the construction of $ACTION$ and following the same procedure as in the proof of Theorem 3.18. ■

Example 3.10 *Let us consider the context free grammar G which generates the arithmetic expressions over the operations $+$, $*$:*

$$E' \to E \quad E \to T + E \,|\, T \quad T \to F * T \,|\, F \quad F \to (\,E\,) \,|\, id$$

It can be checked that G is not a $RL(0)$ grammar, but it is a $SRL(1)$ grammar.

3.2.3 $RL(1)$ grammars

Definition 3.16 *Let $G = (V_N, V_T, S, P)$ be a context free grammar. An $RL(1)$ item for G is a pair $(a, A \to \alpha \centerdot \beta)$, where $A \to \alpha \centerdot \beta$ is an $RL(0)$ item, and $a \in PREVIOUS(A)$ (if $\lambda \in PREVIOUS(A)$, then $a = \#$).*

Definition 3.17 *A* **viable suffix** *for G is a word $\gamma \in V^*$ for which there exists a derivation $S \overset{*}{\underset{lm}{\Longrightarrow}} u\,A\,\alpha \underset{lm}{\Longrightarrow} u\,\beta_1\,\beta_2\,\alpha$, and γ is a suffix for $\beta_1\,\beta_2\,\alpha$. The item $(a, A \to \beta_1 \centerdot \beta_2)$ is called* **valid** *for the viable suffix $\beta_2\,\alpha$ if $a = u^{(1)}$ (if $u = \lambda$ then $a = \#$). The set of all valid items for γ is denoted by $I(\gamma)$.*

Theorem 3.20 *(characterization of $RL(1)$ grammars)*
A reduced grammar $G = (V_N, V_T, S, P)$ is a $RL(1)$ grammar if and only if for any viable suffix γ, there exist no two distinct $LR(1)$ items valid for γ by the form:

$$(a, A \to \centerdot \alpha),\ (b, B \to \beta_1 \centerdot \beta_2\,\gamma) \text{ where } \beta_1^{(1)} \notin V_N \text{ and } a \in LAST(b\,\beta_1)$$

Using a way similar to the $RL(0)$ grammars, for describing the so called $RL(1)$ automaton, we need to give a function called `closure(I)`. This computes all the valid items for the same suffix γ (starting from a given set of valid items).

```
function closure(I):T; /* T is a notation for a subset of valid items */
begin
  I':=I; flag:=true;
  while (flag=true) do begin
    flag:=false;
    for (all (a, A → α B.β) ∈ I') do
      for (all B → γ ∈ P) do
```

```
          for (all b ∈ LAST(a α)) do
            if ((b, B → γ▪) ∉ I') do begin
              I':=I' ∪ {(B → γ▪, b)};
              flag:=true
            end
    end;
    return(I');
  end.
```

Now, we are ready to present an algorithm for constructing the $RL(1)$ automaton for grammar G.

The Algorithm ($RL(1)$-Automaton)

Input: $G = (V_N, V_T, S, P)$ a context free grammar (augmented with the production $S' \to S$);
Output: $M = (T, V, \delta, t_0, T)$ the $RL(1)$ deterministic automaton;
Method:

```
  begin
    t0:=closure((#, S' → S▪)); T:= {t0}; marcat[t0]:=false;
    while (∃t ∈ T and not(marcat[t])) do begin
      for (all X ∈ V) do begin
        t':=∅;
        for (all (a, A → αX▪β) ∈ t) do t':=t' ∪ {(a, A → α▪Xβ)};
        if (t' ≠ ∅) then begin
          t':=closure(t');
          if (t' ∉ T) then begin
            T:=T ∪ {t'};
            marcat[t']:=false;
          end;
          δ'(t, X):=t';
        end
      end;
      marcat[t]:=true;
    end
  end;
```

Lemma 3.4 *The automaton M given in the output of Algorithm ($RL(1)$-Automaton) is deterministic and equivalent (i.e. accepts) to the set of viable suffixes of the input grammar G. Furthermore, for any γ viable suffix, $\delta(t_o, \gamma)$ represents the set of all valid $RL(1)$ items for γ.*

Example 3.11 *The context free grammar given by the productions*

$$S \to R = L \,|\, R,\ L \to R * \,|\, a, R \to L$$

is not a $SRL(1)$ grammar, but it is a $RL(1)$ grammar.

Related to the syntactic algorithm for $RL(1)$ grammars, we can say that it is the same with the corresponding algorithm for $SRL(1)$ grammar. The only possible difference is the way of constructing the $RL(1)$-table.

The Algorithm $RL(1)$-table:

Input: The context free grammar $G = (V_N, V_T, S, P)$, the $RL(1)$ automaton $M = (T, V, \delta, t_0, T)$;
Output: The relation $ACTION(t, a)$, $\forall\ t \in T$, $\forall\ a \in V_T \cup \{\#\}$.
Method:

```
begin
  for (t ∈ T) do begin
    if ((b, A → α a.β) ∈ t) then
      ACTION(t, a):=shift_k, where t_k = δ(t, a);
    if ((a, A → .α) ∈ t) then
      ACTION(t, a):=reduce_r, where r = A → α ∈ P;
    if ((#, A → .α) ∈ t and A ≠ S') then
      ACTION(t, #):=reduce_r, where r = A → α ∈ P;
    if ((#, S' → .S) ∈ t) then
      ACTION(t, #):=ACC_r, where r = S' → S ∈ P;
    for (all a ∈ V_T ∪ {#}) do
      if (ACTION(t, a) = ∅) then ACTION(t, a) = REJ
  end
end.
```

In the next subsection we shall see that in some cases the dimension of the corresponding automaton will be the same as the dimension of the $RL(0)$-automaton. This subclass is called $LARL(1)$ grammars (Look Ahead $RL(1)$).

3.2.4 $LARL(1)$ grammars

In the $RL(1)$ automaton it may happen that some states have the same values on the first component of $RL(1)$ items. So, these states are somehow "equivalent".

Definition 3.18 *Let t be a state from the $RL(1)$ automaton corresponding to the context free grammar G. The* **kernel** *of this state (denoted by $Ker(t)$) is the set of $RL(0)$ items which corresponds to the first component of t, i.e.*
$Ker(t) = \{A \to \alpha_1 . \alpha_2 \mid\ \exists\ (a, A \to \alpha_1 . \alpha_2) \in t\}$.

Example 3.12 $Ker(\{(a, A \to \alpha_1 . \alpha_2),\ (b, A \to \alpha_1 . \alpha_2),\ (c, B \to \beta_1 . \beta_2)\})$ *is the set* $\{A \to \alpha_1 . \alpha_2,\ B \to \beta_1 . \beta_2\}$.

Definition 3.19 *Two states t_1, t_2 of an $RL(1)$ automaton corresponding to a context free grammar G are called* **equivalent** *if they have the same kernel, i.e.* $Ker(t_1) = Ker(t_2)$.

Because every state of a $RL(1)$ automaton is a set of $RL(1)$ items, we may consider the union of two states. Let $t_1 = \{(\{a,b\}, A \to \alpha_1 \centerdot \alpha_2)\}$ and $t_2 = \{(a, A \to \alpha_1 \centerdot \alpha_2)\}$ be two states. Obviously $t_1 \cup t_2 = t_1$ (because $t_2 \subseteq t_1$). If $t_3 = \{(b, A \to \alpha_1 \centerdot \alpha_2)\}$ then $t_2 \cup t_3 = t_1$.

Definition 3.20 *Let $G = (V_N, V_T, S, P)$ be a $RL(1)$ grammar and $M = (T, V, \delta, t_0, T)$ the corresponding $RL(1)$ automaton. We say that G is a $LARL(1)$ grammar if for any pair of equivalent states (t_1, t_2), where $t_1, t_2 \in T$, the state $t_1 \cup t_2$ does not contain conflicts (i.e. reduce - reduce, reduce - shift).*

All the algorithms related to $LARL(1)$ grammars are similar to the corresponding algorithms concerning $RL(1)$ grammars. The only difference concerns the algorithm for constructing the $LARL(1)$ automaton. The following additional comments have to be presented.

First, we compute the $RL(1)$ automaton $M = (T, V, \delta, t_0, T)$ for the $RL(1)$ grammar $G = (V_N, V_T, S, P)$, where for instance $T = \{t_0, t_1, ..., t_n\}$. By determining the equivalent states and making the union operation, we obtain a new set of states, denoted by $T' = \{s_0, s_1, ..., s_m\}$, where $m \leq n$. Now, if T' contains states with conflicts, then we say that G is not a $LARL(1)$ grammar. Otherwise, we compute the automaton $M' = (T', V, \delta', s_0, T')$ in this way. Let s be an arbitrary state which belongs to T'. Then:

- if $s \in T$ then $\delta'(s, X) = \delta(s, X),\ \forall\ X \in V$;
- otherwise $(s \in T' - T)$, $s = t_1 \cup t_2 \cup ... \cup t_k$ $(k \geq 2)$ then $\forall\ X \in V$, the states $\delta(t_1, X)$, $\delta(t_2, X)$, ..., $\delta(t_k, X)$ have the same kernel because t_1, t_2, ..., t_k have the same kernel. Let $s' \in T'$ be the state which has the same kernel as $\delta(t_1, X)$. Now, we define $\delta'(s, X) = s'$.

The $LARL(1)$ table can now be computed with Algorithm $RL(1)$-table, but of course, replacing δ with δ'.

3.3 SIP grammars

The precedence grammars were invented in 1963 by R. W. Floyd ([Flo63]). He defined the main theoretical properties and even a grammar for describing a language closely comparable to ALGOL60. Next, N. Wirth described in 1965 an algorithm for finding precedence functions ([Wir65]). Later, in 1966, N. Wirth and H. Weber described, using precedence grammars, a formal definition for the language Euler - a generalization of ALGOL ([WiW66]). In 1968, N. Wirth described a grammar for PL360 ([Wir68]).

In this section, our intention is to present formal definitions for SIP grammars, which can be viewed as "mirroring" the precedency grammars.

Definition 3.21 *Let $G = (V_N, V_T, S, P)$ be a context free grammar without null productions. We consider the following binary relations $<\cdot$, $\doteq\ \subseteq V \times V$ and $\cdot> \subseteq V \times V_T$ as:*

- $X \lessdot Y$ *if there exists a production* $A \to \alpha\, X\, B\, \beta \in P$ *and* $B \stackrel{+}{\Longrightarrow} Y\, \gamma$;
- $X \doteq Y$ *if there exists a production* $A \to \alpha\, X\, Y\, \beta \in P$;
- $X \gtrdot a$ *if there exists a production* $A \to \alpha\, B\, Y\, \beta \in P$, $B \stackrel{+}{\Longrightarrow} \gamma\, X$ *and* $Y \stackrel{*}{\Longrightarrow} a\, \delta$.

Definition 3.22 *A context free grammar G without null productions in which the binary relations* $\lessdot$, $\doteq$, $\gtrdot$ *are disjoint, is called a* **precedence grammar**. *Furthermore, if G satisfies the statement (***invertible grammar***):*

$$\forall\ A \to \beta \in P,\ \forall A \to \beta' \in P \Longrightarrow A = A'$$

then G is called a **simple precedence grammar** *(denoted by SP grammar).*

Definition 3.23 *We say that a context free grammar G is an inverse precedence grammar if* $\widetilde{G}$ *is a precedence grammar. If G is invertible, then G is called a* **simple inverse precedence grammar** *(denoted by SIP grammar).*

Definition 3.24 *Let* $G = (V_N, V_T, S, P)$ *be a context free grammar without null productions. We consider the following relations (called* **inverse precedence relations***)* $<\!\!<\!\cdot \subset V_T \times V$ *and* $\dot{\equiv}$, $\cdot\!>\!\!> \subseteq V \times V$ *as:*

- $a <\!\!<\!\cdot X$ *if there exists a production* $A \to \alpha\, B\, C\, \beta \in P$, $B \stackrel{*}{\Longrightarrow} \gamma\, a$ *and* $C \stackrel{+}{\Longrightarrow} X\, \delta$;
- $X \dot{\equiv} Y$ *if there exists a production* $A \to \alpha\, X\, Y\, \beta \in P$;
- $X \cdot\!>\!\!> Y$ *if there exists a production* $A \to \alpha\, B\, Y\, \beta \in P$ *and* $B \stackrel{+}{\Longrightarrow} \gamma\, X$;

As an extension to a special terminal symbol #, we may say that $\# <\!\!<\!\cdot X$ iff $\exists\ S \stackrel{+}{\Longrightarrow} X\, \alpha$ and $X \cdot\!>\!\!> \#$ iff $\exists\ S \stackrel{+}{\Longrightarrow} \alpha\, X$.

According to Definitions 3.21, 3.23 and 3.24, we can say that the following statements hold:

- $X \lessdot Y$ in G iff $Y \cdot\!>\!\!> X$ in $\widetilde{G}$;
- $X \doteq Y$ in G iff $Y \cdot\!>\!\!> X$ in $\widetilde{G}$;
- $X \gtrdot Y$ in G iff $Y <\!\!<\!\cdot X$ in $\widetilde{G}$.

Therefore, a context free grammar G without null productions, in which the binary relations $<\!\!<\!\cdot$, $\dot{\equiv}$, $\cdot\!>\!\!>$ are disjoint, is called an **inverse precedence grammar.**

Theorem 3.21 *Let* $G = (V_N, V_T, S, P)$ *be a reduced context free grammar without null productions, let*

$$\#S\# \underset{lm}{\stackrel{*}{\Longrightarrow}} u_1\, u_2 \dots u_k\, A\, X_1\, X_2 \dots X_n \underset{lm}{\Longrightarrow} u_1\, u_2 \dots u_k\, Y_1\, Y_2 \dots Y_m\, X_1\, X_2 \dots X_n$$

be an arbitrary derivation for which $u_1 = \#$, u_2, ..., $u_k \in V_T$, $X_n = \#$. *Then the following statements are fulfilled:*

1. $u_k \lessdot\!\!\lessdot Y_1$;
2. $Y_k \doteq Y_{k+1}, \ \forall\ k = \overline{1, m-1}$;
3. $Y_m \cdot\!\!>\!\!> X_1$;
4. $X_k \cdot\!\!>\!\!> X_{k+1}$ *or* $X_k \doteq X_{k+1}, \ \forall\ k = \overline{1, n-1}$.

Proof By induction on the number of derivation steps. ∎

Definition 3.25 *Let* $G = (V_N, V_T, S, P)$ *be an inverse precedence grammar and* $<<\cdot$, $\doteq$, *and* $\cdot>>$ *the corresponding inverse precedence relations. We denote by* $\mathcal{C} \subseteq \# V_T^* \times V^* \# \times \{1, 2, ..., |P|\}^* \cup \{ACC, REJ\}$ *the set of all possible configurations, where* $\#$ *is a special character (a new terminal symbol). The* **simple inverse precedence parser** *(denoted by* $SIPP(G)$*) is the pair* $(\mathcal{C}_0, \vdash)$*, where the set* $\mathcal{C}_0 = \{(\#w, \#, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ *is called the set of* **initial configurations**, *and* $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ *is the* **transition relation** *(sometimes denoted by* $\underset{SIPP(G)}{\longmapsto}$*) between configurations given by:*

1^0 *Shift:* $(\#u\,a, \gamma\#, \pi) \vdash (\#u, a\,\gamma\#, \pi)$ *if* $a \cdot\!\!>\!\!>^{(1)} \gamma$ *or* $a \doteq^{(1)} \gamma$;

2^0 *Reduce:* $(\#u, \beta\,\gamma\#, \pi) \vdash (\#u, A\,\gamma\#, r\,\pi)$ *if* $\beta = \beta_1 \dots \beta_m$, $u^{(1)} <<\cdot \beta_1$, $\beta_k \doteq \beta_{k+1}, \ \forall\ k = \overline{1, m-1}$, $\beta_m \cdot\!\!>\!\!>^{(1)} \gamma$, $r = A \to \beta$;

3^0 *Accept:* $(\#, S\#, \pi) \vdash ACC$;

4^0 *Reject:* $(\#u, \gamma\#, \pi) \vdash REJ$ *if no transitions of type* 1^0, 2^0 *and* 3^0 *can be applied.*

The parser $SIPP(G)$ is similar to the parser $PRL_0(G)$, the only difference being the auxiliary parsing informations (the relations $<<\cdot$, $\doteq$, $\cdot>>$).

Example 3.13 *Let* G *be a context free grammar giben by the productions:*

$$S \to a\,S\,S\,b \mid c$$

By computing the binary relations $<<\cdot$, $\doteq$, $\cdot>>$, *we obtain:*

- $<<\cdot = \{(\#, a),\ (\#, c),\ (a, a),\ (a, c),\ (b, c),\ (b, a),\ (c, a),\ (c, c)\}$;
- $\doteq = \{(a, S),\ (S, S),\ (S, b)\}$;
- $\cdot>> = \{(b, \#),\ (c, \#),\ (b, b),\ (b, S),\ (c, b),\ (c, S)\}$.

Therefore, because the relations $<<\cdot$, $\doteq$, $\cdot>>$ *are disjoint, we can say that* G *is an inverse precedence grammar. Because it is invertible, too,* G *is a simple inverse precedence grammar.*

Now, let $w = a\,a\,c\,c\,b\,c\,b$ *be a input word for the parser* $SIPP(G)$*. We obtain the transitions array:*

$(\#a\,a\,c\,c\,b\,c\,b, \#, \lambda) \overset{1^0}{\vdash} (\#a\,a\,c\,c\,b\,c, b\#, \lambda) \overset{1^0}{\vdash} (\#a\,a\,c\,c\,b, c\,b\#, \lambda) \overset{2^0}{\vdash}$
$(\#a\,a\,c\,c\,b, S\,b\#, [2]) \overset{1^0}{\vdash} (\#a\,a\,c\,c, b\,S\,b\#, [2]) \overset{1^0}{\vdash} (\#a\,a\,c, c\,b\,S\,b\#, [2]) \overset{2^0}{\vdash}$
$(\#a\,a\,c, S\,b\,S\,b\#, [2,2]) \overset{1^0}{\vdash} (\#a\,a, c\,S\,b\,S\,b\#, [2,2]) \overset{2^0}{\vdash} (\#a\,a, S\,S\,b\,S\,b\#, [2,2,2]) \overset{1^0}{\vdash}$
$(\#a, a\,S\,S\,b\,S\,b\#, [2,2,2]) \overset{2^0}{\vdash} (\#a, S\,S\,b\#, [1,2,2,2]) \overset{1^0}{\vdash} (\#, a\,S\,S\,b\#, [1,2,2,2]) \overset{2^0}{\vdash}$
$(\#, S\#, [1,1,2,2,2]) \overset{3^0}{\vdash} ACC.$

Therefore $w \in L(G)$, *and* w *has the left syntactic analysis* $\pi_l = [1,1,2,2,2]$.

Now, we present the parsing algorithm for simple inverse precedence grammar. We consider the predefined procedures **pop**, **push** already known.

The Algorithm (Parsing-SIP)

Input: The SIP grammar $G = (V_N, V_T, S, P)$, the binary relations $<\!\!<\cdot$, $\doteq$, $\cdot\!>\!\!>$ and the word $w \in V_T^*$;
Output: The left syntactic analysis if $w \in L(G)$; otherwise the message: 'w is not accepted'.
Method:
The main program is:

```
begin
   read(w);push(stack,#);i:=|w|+1;
   accept:=false;is_over:=false;
   put in the input tape the string "#w";
   repeat
     pop(stack,Y1);
     if (i>1) and ((w[i]·>>Y1) or (w[i]≐Y1 )) then begin
       /* shift action */
       push(stack,w[i]);
       i:=i-1;
     end
     else
       if (i=1) then begin
         if (stack="#") then
           /* accept action */
           accept:=true
         else /* reject action */
           is_over:=true;
       end
       else
         if (w[i]<<·Y1) then begin
           let Y1 Y2 ... Ym X1 be the string from the stack for which:
           Yk≐Yk+1, ∀ i = 1,m-1 (overlined), Ym≐X1;
           find a production of the form r= A → Y1 Y2 ... Ym ∈ P;
           if (does not exist such a production) then
             /* reject action */
             else is_over:=true
           end
           else begin
             /* reduce action */
             /* r is unique because G is invertible */
             remove the string Y1 Y2 ... Ym from the stack;
             push(stack,A);
             push(Output_tape,r)
           end
         end
         else is_over:=true /* reject action */
```

```
   until (is_over=true);
   if (accept=true) then
     write('w is accepted and has the left syntactic analysis ', Output_tape)
   else write('w is not accepted.')
end.
```

Theorem 3.22 *(correctness and complexity of Algorithm (Parsing-SIP))*
Algorithm (Parsing-SIP) is correct and has the time complexity of $\mathcal{O}(|w|)$*, where* w *is the input word.*

Proof Follows the same lines as the Proof of Theorem 3.18. ∎

4 Deterministic bidirectional parsing for context free languages

In this section, we shall present how we can combine some subclasses of context free grammars to obtain deterministic (and linear) parallel algorithms for solving the membership problem.

The deterministic bidirectional parsers have the same device as the general model, the only difference being the uniqueness of choosing the production r from the set of productions of the input grammar.

Next, we shall define 10 subclasses of context free grammars.

Definition 4.1 *Let* G *be a context free grammar and* $k \in \mathbf{N}$*. We say that:*

1. G *is a* $LL(k) - RL(0)$ *grammar if* G *is a* $LL(k)$ *and* $RL(0)$ *grammar;*
2. G *is a* $LL(k) - SRL(1)$ *grammar if* G *is a* $LL(k)$ *and* $SRL(1)$ *grammar;*
3. G *is a* $LL(k) - RL(1)$ *grammar if* G *is a* $LL(k)$ *and* $RL(1)$ *grammar;*
4. G *is a* $LL(k) - LARL(1)$ *grammar if* G *is a* $LL(k)$ *and* $LARL(1)$ *grammar;*
5. G *is a* $LL(k) - SIP$ *grammar if* G *is a* $LL(k)$ *and* SIP *grammar;*
6. G *is a* $LR(0) - RR(k)$ *grammar if* G *is a* $LR(0)$ *and* $RR(k)$ *grammar;*
7. G *is a* $SLR(1) - RR(k)$ *grammar if* G *is a* $SLR(1)$ *and* $RR(k)$ *grammar;*
8. G *is a* $LR(1) - RR(k)$ *grammar if* G *is a* $LR(1)$ *and* $RR(k)$ *grammar;*
9. G *is a* $LALR(1) - RR(k)$ *grammar if* G *is a* $LALR(1)$ *and* $RR(k)$ *grammar;*
10. G *is a* $SP - RR(k)$ *grammar if* G *is a* SP *and* $RR(k)$ *grammar.*

Of course, we can easily extend the above definition to the languages. For instance, we say that L is a $LL(k) - RL(0)$ language if there exists $k \in \mathbf{N}$ and G a $LL(k) - RL(0)$ grammar such that $L = L(G)$.

Corollary 4.1 *The following statements are fulfilled:*

1. G *is a* $LL(k) - RL(0)$ *grammar iff* $\widetilde{G}$ *is a* $LR(0) - RR(k)$ *grammar;*
2. G *is a* $LL(k) - SRL(1)$ *grammar iff* $\widetilde{G}$ *is a* $SLR(1) - RR(k)$ *grammar;*
3. G *is a* $LL(k) - RL(1)$ *grammar iff* $\widetilde{G}$ *is a* $LR(1) - RR(k)$ *grammar;*
4. G *is a* $LL(k) - LARL(1)$ *grammar iff* $\widetilde{G}$ *is a* $LALR(1) - RR(k)$ *grammar;*
5. G *is a* $LL(k) - SIP$ *grammar iff* $\widetilde{G}$ *is a* $SP - RR(k)$ *grammar;*

Proof By Definitions 3.2, 3.9, 3.23 and 4.1. ■

It is obvious that all the languages associated to the grammars presented in Definition 4.1 are deterministic context free languages. Using a "mirroring" strategy, we can easily extend from the literature ([AhU72], [Har78], [HoU79], [Sal73]) the specific results for inclusions, hierarchies of classical subclasses of deterministic context free languages. Therefore, the following relations hold ($\forall\ k \in \mathbf{N}$):

- $RL(k) = RL(1)$, $RR(k) \subseteq RL(k)$, $SIP \subseteq RL(1)$, $RR(k) \subseteq RR(k+1)$;
- $LL(k) - RL(0) \subseteq LL(k) - SRL(1) \subseteq LL(k) - RL(1)$;
- $LL(k) - RL(0) \subseteq LL(k) - LARL(1) \subseteq LL(k) - RL(1)$;
- $LR(0) - RR(k) \subseteq SLR(1) - RR(k) \subseteq LR(1) - RR(k)$;
- $LR(0) - RR(k) \subseteq LALR(1) - RR(k) \subseteq LR(1) - RR(k)$.

We note that the first five classes of grammars (Definition 4.1) use a left bidirectional strategy and the last five use a right bidirectional strategy.

In particular, this deterministic parallel approach is very similar to the general parallel aproach, the only difference is the missing of backtrack steps. Thus the kernel of the parallel iteration corresponding to the left bidirectional strategy is (we use the same considerations of Algorithm (PAR_LEFT)):

```
repeat in parallel
  if (i1<=i2) then action1(P1);
  action2(P2);
until (i1>=i2) or (exit=true);
```

where `action1`, respectively `action2`, are procedures related to sequential algorithms for syntactic analysis (i.e. associated to the classical subclasses of grammars and to the subclasses described in Section 3, such as Algorithms Parsing-RR(1), Parsing-RL(0), Parsing-SLR(1), Parsing-SIP). But this time, instead of exponential sequential running time, we have a linear running time for the procedures `action1` and `action2`, because backtrack steps are not necessary. The linear time complexity follows from Theorems 3.12, 3.18, 3.19, 3.22 and the similar classical results.

The correctness of the deterministic parallel algorithms is ensured by the correctness of the general parallel algorithm and the correctness of each of the sequential syntactic analysers for our subclasses of context free grammars.

Theorem 4.1 *(the complexity of the deterministic parallel algorithms)*

Let us denote with $T_1(n)$, $T_2(n)$ the running time of the sequential syntactic analysers from Section 3, where n is the length of the input word. Then the parallel running time $t(n)$ satisfies the relations:

- $\frac{\min\{T_1(n),T_2(n)\}}{2} + K \leq t(n) \leq \max\{T_1(n), T_2(n)\}$. *(we suppose that the time routing is zero and K is a constant not depending on n);*
- $t(n) \in \mathcal{O}(n)$.

Proof The inequality $t(n) \leq \max\{T_1(n), T_2(n)\}$ can be obtained by supposing that one processor stays. For instance, if `P1` stays, then $t(n) = T_2(n)$ (time routing is zero).

The other inequality can be obtained by supposing that both processors work until $i1 = i2$. This means a running time of $\frac{\min\{T_1(n),T_2(n)\}}{2}$. Then one processor stays and the other (possibly) performs some constant number of iterations.

The fact that $t(n) \in \mathcal{O}(n)$ is obvious because of the linear complexity for the deterministic parsers associated to our subclasses of grammars. ■

Example 4.1 *Let us consider the context free grammar*

$$G = (\{S', S, B, C\}, \{a, b, e, ;\}, S', P)$$

where the set of productions P is:

1. $S' \to S$ 2. $S \to \lambda$ 3. $S \to B$ 4. $B \to a$
5. $B \to b\,S\,C\,e$ 6. $C \to \lambda$ 7. $C \to ;\,S\,C$

We shall see that G is a $LL(1) - LARL(1)$ grammar. First, we compute the following sets:

X	S'	S	B	C
$FIRST$	$\{a, b, \lambda\}$	$\{a, b, \lambda\}$	$\{a, b\}$	$\{;, \lambda\}$
$FOLLOW$	$\{\lambda\}$	$\{e, ;, \lambda\}$	$\{e, ;\}$	$\{e\}$

Now, we compute the sets of "lookahead" symbols and thus we can check whether G is a $LL(1)$ grammar or not:

- $FIRST(FOLLOW(S)) = \{e, ;, \lambda\}$, $FIRST(B\,FOLLOW(S)) = \{a, b\}$;
- $FIRST(a\,FOLLOW(B)) = \{a\}$, $FIRST(b\,S\,C\,e\,FOLLOW(B)) = \{b\}$;
- $FIRST(FOLLOW(C)) = \{e\}$, $FIRST(;\,S\,C\,FOLLOW(C)) = \{;\}$.

Because any two of these sets are disjoint, it follows that G is a $LL(1)$ grammar. For testing the $LARL(1)$ property, we need some auxiliary informations, such as the sets $LAST$ and the $RL(1)$ automaton attached to G.

X	S'	S	B	C
$LAST$	$\{a, e, \lambda\}$	$\{a, e, \lambda\}$	$\{a, e\}$	$\{a, e, ;, \lambda\}$

Now, we are ready to construct the $RL(1)$ automaton for G.

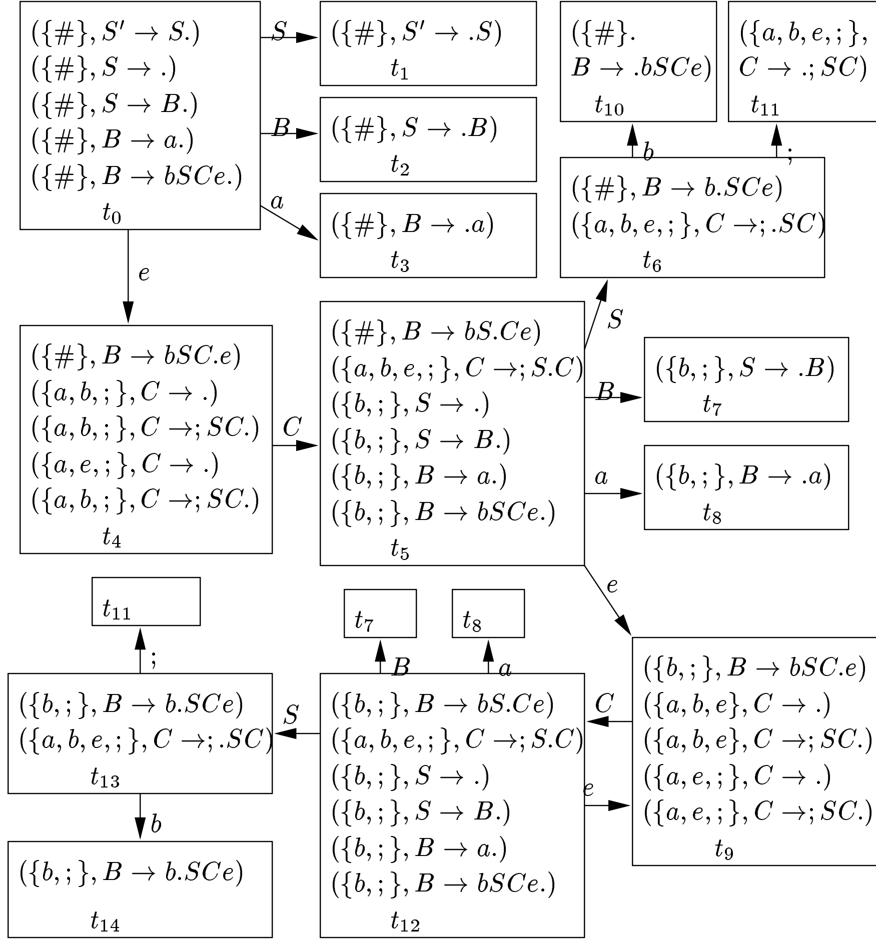


Figure 9. $RL(1)$ automaton for G

In Figure 9, the arcs (t_{12}, t_7), (t_{12}, t_8) *and* (t_{13}, t_{11}) *are pointed out in a different way because of the picture size.*

It can be checked on the $RL(1)$ *automaton that* G *is a* $RL(1)$ *grammar according to Theorem 3.20. Furthermore, the following pairs of states are equivalent (Definition 3.19):*

$$(t_2, t_7),\ (t_3, t_8),\ (t_4, t_9),\ (t_5, t_{12}),\ (t_6, t_{13}),\ (t_{10}, t_{14})$$

Therefore, the $LARL(1)$ *automaton will have only* 9 *states, i.e. the set of states will be* $\{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_{10}, t_{11}\}$. *Because the* $LARL(1)$ *automaton has no conflicts in its states, we conclude that* G *is a* $LARL(1)$ *grammar. Denoting* $reduce_r$ *by* R_r *and* $shift_k$ *by* S_k, *the associated* $LARL(1)$ *table will be:*

ACTION	#	*a*	*b*	*e*	;	δ	*S*	*B*	*C*
t_0	R_2	S_3		S_4		t_0	t_1	t_2	
t_1	R_1					t_1			
t_2	R_3		R_3		R_3	t_2			
t_3	R_4		R_4		R_4	t_3			
t_4		R_6	R_6	R_6	R_6	t_4			t_5
t_5		S_3	R_2	S_4	R_2	t_5	t_6	t_2	
t_6			S_{10}		S_{11}	t_6			
t_{10}	R_5		R_5		R_5	t_{10}			
t_{11}		R_7	R_7	R_7	R_7	t_{11}			

The empty places in the above table mean REJ configuration, i.e. the rejection of the input word.

Let us consider the input word $w = b\,a\,;\,b\,a\,e\,e$. *We shall present the transitions for the deterministic left bidirectional parser of the corresponding* $LL(1) - LARL(1)$ *grammar. In the following, we shall suppose that the processors operate in a synchronous way, i.e. the processor P1 waits for the termination of the operations from P2, and viceversa. In that case, we present a possible parallel running of that two processors.*

Step	*Action1*	`Output_tape1`	`Stack1`	`i1`
0.	*Initial*	λ	S'	1
1.	*Expand*	[1]	S	1
2.	*Expand*	[1, 3]	B	1
3.	*Expand*	[1, 3, 5]	$b\,S\,C\,e$	1
4.	*Reduce*	[1, 3, 5]	$S\,C\,e$	2
5.	*Expand*	[1, 3, 5, 3]	$B\,C\,e$	2
6.	*Expand*	[1, 3, 5, 3, 4]	$a\,C\,e$	2
7.	*Reduce*	[1, 3, 5, 3, 4]	$C\,e$	3
8.	*Expand*	[1, 3, 5, 3, 4, 7]	$;\,S\,C\,e$	3

Step	*Action2*	`i2`	`Stack2`	`Output_tape2`
0.	*Initial*	7	t_0	λ
1.	*Shift*	6	$t_4\,e\,t_0$	λ
2.	*Reduce*	6	$t_5\,C\,t_4\,e\,t_0$	[6]
3.	*Shift*	5	$t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[6]
4.	*Reduce*	5	$t_5\,C\,t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[6, 6]
5.	*Shift*	4	$t_3\,a\,t_5\,C\,t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[6, 6]
6.	*Reduce*	4	$t_2\,B\,t_5\,C\,t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[4, 6, 6]
7.	*Reduce*	4	$t_6\,S\,t_5\,C\,t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[3, 4, 6, 6]
8.	*Shift*	3	$t_{10}\,b\,t_6\,S\,t_5\,C\,t_4\,e\,t_5\,C\,t_4\,e\,t_0$	[3, 4, 6, 6]
9.	*Reduce*	3	$t_2\,B\,t_5\,C\,t_4\,e\,t_0$	[5, 3, 4, 6, 6]
10.	*Reduce*	3	$t_2\,S\,t_5\,C\,t_4\,e\,t_0$	[3, 5, 3, 4, 6, 6]
11.	*Shift*	2	$t_{11}\,;\,t_2\,S\,t_5\,C\,t_4\,e\,t_0$	[3, 5, 3, 4, 6, 6]

The processor P1 is waiting the last three steps of the processor P2 are executed. The test "`if (Stack1=Stack2) then`*" from the general left bidirectional algorithm (PAR_LEFT) has to be view as "*`if (Stack1=`h`(Stack2)) then`*",*

where
$h_1 : V \cup T \to V$ *given by:*

$$h_1(X) = \begin{cases} X & \textit{if } X \in V \\ \lambda & \textit{otherwise} \end{cases}$$

Of course, we can extend it to the words of arbitrary length using the function $h : (V \cup T)^* \to V^*$, *given by:*

$$h(\lambda) = \lambda, \quad h(X_1 \dots X_n) = h_1(X_1) \cdot \dots \cdot h_1(X_n).$$

We remind the reader that T *is the set of states from the* $RL(1)$ *automaton. Now, it is obvious that*

$$h(\texttt{Stack2}) = h(t_{11} \,;\, t_2\, S\, t_5\, C\, t_4\, e\, t_0) =;\, S\, C\, e = \texttt{Stack1}$$

Therefore, the word w *is accepted by the parallel algorithm and has the left syntactic analysis* $[1, 3, 5, 3, 4, 7, 3, 5, 3, 4, 6, 6]$.

5 Conclusions

As it was also described in Example 1.2 the main complexity result of our paper is Theorem 4.1.

We think that the concept of bidirectional parsing for context free grammars described in this paper will contribute to a new view for describing compilers on computers with two processors.

Open problems:

- to find new subclasses of deterministic parallel algorithms for simulating the bidirectional parsing;
- to estimate more precisely the running time of the deterministic parallel algorithm presented in Section 4;
- to find further closure properties of the subclasses of the described languages.

References

[AnK98] Andrei, Şt., Kudlek, M.: Linear Bidirectional Parsing for a Subclass of Linear Languages. B-215, Fachbereich Informatik, Universität Hamburg, pp. 1-20 (1998)

[AhU72] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Volume I, II, Prentice Hall, 1972

[Akl97] Akl, S.: Parallel Computation. Models and Methods. Prentice Hall, 1997

[AnG95] Andrei, Şt., Grigoraş, Gh.: Tehnici de compilare. Lucrări de laborator. Editura Universităţii "Al.I.Cuza", Iaşi, 1995

[Flo63] Floyd, R.W.: Syntactic Analysis and Operator Precedence. Journal of the ACM 10:3, pp. 316-333 (1963)

[GiR89] Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambrigde University Press, 1989

[Gri86] Grigoraş, Gh.: Limbaje formale şi tehnici de compilare. Editura Universităţii "Al.I.Cuza", Iaşi, 1986

[Har78] Harrison, M. A.: Introduction to Formal Language Theory. Addison - Wesley Publishing Company, 1978

[Hay88] Hayes, J.P.: Computer Architecture and Organization, McGraw-Hill International Editions, 1988

[HoU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison - Wesley Publishing Company, 1979

[Knu65] Knuth, D.E.: On the translation of languages from left to right. Information Control, No. 8, pp. 607-639 (1965)

[Knu71] Knuth, D.E.: Top-down analysis. Acta Informatica. No. 1, pp. 79-110 (1971)

[JuA97] Jucan, T, Andrei, Şt.: Limbaje formale şi teoria automatelor. Culegere de probleme. Editura Universităţii "Al. I. Cuza", Iaşi, 1997

[LeS68] Lewis, P.M., Stearns, R.: Syntax-directed transduction. Journal of the ACM. 15, pp. 464-488 (1968)

[Sal73] Salomaa, A.: Formal Languages. Academic Press. New York, 1973

[Wir65] Wirth, N.: Algorithm 265: Find Precedence Functions. Comm. ACM 8:10, pp. 604-605 (1965)

[Wir68] Wirth, N.: PL360 - a programming language for the 360 computers. Journal of the ACM 1: pp. 37-44 (1968)

[WiW66] Wirth, N., Weber, H.: Euler - a generalization of Algol and its formal definition. Parts 1 and 2. Comm. ACM 9: pp. 1-2, 13-23 89-99 (1966)