BIDIRECTIONAL PARSING

DISSERTATION

ZUR **E**RLANGUNG DES **D**OKTORGRADES
AM **F**ACHBEREICH **I**NFORMATIK
DER **U**NIVERSITÄT **H**AMBURG

vorgelegt von

Ştefan Andrei
geboren in Buhuşi, Rumänien

Hamburg, im Februar 2000

Genehmigt vom Fachbereich Informatik der Universität Hamburg
auf Antrag von Prof. Dr. Manfred Kudlek (Betreuer)
und Prof. Dr. Cristian Masalagiu (Gutachter)
und Dr. Martin Lehmann. (Gutachter)

Hamburg, den 21. Februar 2000 (Datum der Disputation)

Prof. Dr. Leonie Dreschler-Fischer
Dekanin des Fachbereichs Informatik

Contents

1 Introduction **1**
- 1.1 Basic notions . . . 1
- 1.2 The structure of the thesis . . . 2
- 1.3 Acknowledgments . . . 3

2 Preliminaries **5**
- 2.1 Context free grammars . . . 5
  - 2.1.1 Definitions and notations . . . 5
  - 2.1.2 Some properties of context free and linear languages . . . 9
- 2.2 Attribute grammars . . . 11
  - 2.2.1 Definitions, notations, examples . . . 12
  - 2.2.2 The circularity problem . . . 19
  - 2.2.3 Transitive and reflexive closure . . . 24
- 2.3 Parallel computers . . . 29
  - 2.3.1 Models of (parallel) computation . . . 29
  - 2.3.2 Analyzing algorithms . . . 35

3 Linear-time bidirectional parsing for a subclass of linear grammars **39**
- 3.1 Definitions and general properties . . . 39
- 3.2 A bidirectional parser for $LLin(m, n)$ grammars . . . 45
- 3.3 Bidirectional parsing for $LLin(1, 1)$ grammars . . . 50
- 3.4 Conclusions . . . 54

4 Left and right bidirectional parsing for context free grammars **57**
- 4.1 Left and right bidirectional parsers . . . 57
- 4.2 A parallel approach for (general) bidirectional parsing . . . 70
- 4.3 Deterministic subclasses of context free grammars . . . 76
  - 4.3.1 $RR(k)$ grammars . . . 77
  - 4.3.2 $RL(k)$ grammars . . . 83
  - 4.3.3 SIP grammars . . . 98
- 4.4 Deterministic bidirectional parsing for context free languages . . . 101
- 4.5 Conclusions . . . 106

5 Up-to-up bidirectional parsing for context free grammars 107
5.1 The general up-to-up bidirectional parser 107
5.2 Parallel approach for general up-to-up bidirectional parsing . . . 120
5.3 Deterministic up-to-up bidirectional parsing for context free languages 126
5.4 Conclusions 128

6 Bidirectional attribute evaluation 129
6.1 Data representations of trees and their bidirectional traversal . . 129
6.2 Bidirectional attribute evaluation 133
6.3 Conclusions 136

7 Final conclusions 139
7.1 Original contributions of this thesis 139
7.2 Future work 140

Bibliography 141

Chapter 1

Introduction

1.1 Basic notions

The syntax of programming languages is usually expressed using context free grammars.

The known membership problem for a context free language (given by a context free grammar) includes a recognition side (syntactic analysis). The *parser* yields not only a *yes* or *no* answer, but also a *parse tree*, which describes the whole history of the (possible) acceptance.

The theory of recognition and parsing for context free grammars dates back to the sixties. Existing literature can be grouped into at least two areas: *theoretical complexity of recognition and parsing* and *parsing of programming languages*.

The main concern of the first area is the space and time complexity of recognition and parsing. The underlying machine model is an abstract device such as a pushdown automaton or a Turing machine ([AhU72, ASU86, Har78, HoU79, Sal73, WaG84]). The class of context free grammars under consideration may be considerably large (deterministic, unambiguous, or even general context free grammars) or particularly small (bracket or input-driven grammars).

The parsers are used as parts of compilers. Therefore, they must be efficient. Most parsing algorithms are of linear time and space complexity. This however requires a severe restriction on the class of possible input context free grammars. This has led to a wide variety of grammar subclasses, such as $LL(k)$, $LR(k)$, $LC(k)$, precedence grammars, etc. ([AhU72]). Generally, a parser is built for one specific context free (class of) grammar(s), rather than for all possible ones. Thus, a context free grammar may also be seen as a specification of a parser and parsers can be often automatically derived from given context free grammars by means of a parser generator.

There exists a generally advocated and well understood theory for sequential parsing of programming languages. In sequential parsing, efficiency generally implies linear time complexity. Parallel algorithms - even they may not decrease

the efficiency of the given algorithms - are useful for a clear understanding of the underlying problem. Anyway, in order to have better results, our bidirectional approach is a good and necessary step.

1.2 The structure of the thesis

This thesis deals with parallel algorithms for parsing and attribute evaluation. We describe some subclasses of context free grammars for which a parallel approach useful for solving the membership problem is defined. More precisely, we have combined the classical type of parser attached to a grammar G with a "mirror" one. We have called what we get a *bidirectional parser* because it analyses the input word from both sides using two processors.

The first section of Chapter 2 presents basic notions about context free grammars (definitions, notations, properties, important subclasses) taken from [HoU79] and [Sal73].. The next section does the same for attribute grammars ([Alb91a, And97]). The transitive (and reflexive) closure of a graph has also been presented in this chapter.The last section is an introduction to models of parallel (SISD, MISD, SIMD, MIMD) computers and to the analysis of parallel algorithms ([Akl97]).

Chapter 3, called *Linear-time bidirectional parsing for a subclass of linear languages*, describes new subclasses of linear grammars, denoted by $LLin(m,n)$, $m,n \in \mathbf{N}$ ([AnK98, AnK99a]). These are similar to the classical $LL(k)$, $k \in \mathbf{N}$ ([AhU72, LeS68]) grammars. Intuitively, *looking ahead* to the next m terminal symbols and *looking back* to the previous n terminal symbols suffices to uniquely determine the production which has to be applied. The membership problem for $LLin(m,n)$ grammars can be solved using a linear time complexity algorithm.

Chapter 4 (*Left and right bidirectional parsing for context free grammars*) presents **left and right bidirectional parser** for general (and some subclasses of) context free languages ([AnK99b]). Deterministic subclasses of context free grammars (such as $RR(k)$, $RL(k)$, SIP grammars) are used for obtaining ten combinations of new subclasses. The membership problem for all these types of grammars can be solved with a parallel algorithm in linear time using two processors.

In Chapter 5 (*Up-to-up bidirectional parsing for context free grammars*), we introduce the **up-to-up bidirectional parser** ([AGK99]). It combines the deterministic subclasses ($LR(k)$, $RL(k)$) with the up-to-up bidirectional strategy. The membership problem can now be solved in linear time complexity with a parallel two processors algorithm.

Chapter 6, called *Bidirectional attribute evaluation*, describes a parallel two processors algorithm for evaluating the attribute instances of an attributed derivation tree. We have called this strategy the **bidirectional attribute evaluation** ([AKM99]).

Final conclusions (Chapter 7) and the Bibliography end this thesis.

1.3 Acknowledgments

First, I want to express my gratitude for the extremely competent guidance of Prof. Dr. Manfred KUDLEK, Fachbereich Informatik, Universität Hamburg, who has accepted to be my doctoral professor advisor. There are many other persons which had important remarks, among them being Prof. Dr. Martin LEHMANN. During my staying in Hamburg - not possible without the financial support of Joint Japan/World Bank Graduate Scholarship Program - I have had many useful discussions with Assistant Olaf KUMMER who has also helped me to pass over some difficult moments.

Next, my thanks go to the teachers of the Faculty of Computer Science, "Al.I.Cuza" University and especially to Prof. Dr. Toader JUCAN (for a permanent stimulation of my training and research activity), Prof. Dr. Gheorghe GRIGORAŞ (who cultivated my interest in compiling and - mainly - in parsing, beginning with the college days) and to Prof. Dr. Cristian MASALAGIU (for his carefully and expertly comments on my entire work and results).

I am also grateful to my wife Liliana and my daughter Anca who have had a particular patience and understanding for my frequent and long home absences.

My father and mother have always encouraged me to learn and finally obtain a Ph. D. degree. I have also to thank to my father and mother-in-low who have taken care of my family life and needs.

Chapter 2

Preliminaries

This chapter presents basic notions (definitions, notations, properties, important subclasses) about context free grammars, attribute grammars, closure of a graph, models of parallel computers and analysis of parallel algorithms.

Let us fix some notations to express lower and upper bounds of the number of steps required for solving a problem in the worst case. Let f and g 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 $n \geq n_0$.

2.1 Context free grammars

2.1.1 Definitions and notations

The term **grammar** was firstly used by the linguist and philosopher Noam Chomsky for defining the generating systems ([Cho56, Cho59]). *Context free grammars* are one of the basic notions in compiling theory.

Definition 2.1.1 *An* **alphabet** *is a nonempty, finite set (of* **symbols***). A* **word over** *an alphabet V is an application $p : \{1, 2, ..., n\} \to V$, $n = |p|$ ($n \in \mathbf{N}$) being the length of the word p.*

Notation 2.1.1 *We denote by $V^n = \{p \mid p : \{1, 2, ..., n\} \to V\}$ the set of all words over V of length n. The empty word (of length 0) is denoted by λ. Furthermore, we denote $V^* = \bigcup_{n \geq 0} V^n$ and $V^+ = \bigcup_{n \geq 1} V^n$.*

Definition 2.1.2 *$\mathcal{G} = (\mathcal{V}, E, s, d)$ is an* **oriented graph** *if $\mathcal{V}$ is a nonempty set of vertices, E is a nonempty set of edges and $s : E \to \mathcal{V}$, $d : E \to \mathcal{V}$ are the source and the destination functions. If $s(e) = v$ and $d(e) = v'$, then the edge e will be denoted $e : v \to v'$ or $v \stackrel{e}{\to} v'$, or simply (v, v') if the name is not important (i.e. there exists - at least - one edge from v to v').*

For any $v \in \mathcal{V}$, let $E(v) = \{e \mid s(e) = v\}$ and $S(v) = \{v' \mid \exists\ e \in E$ such that $v \stackrel{e}{\to} v'\}$.

Definition 2.1.3 *For an oriented graph $\mathcal{G} = (\mathcal{V}, E, s, d)$, if $\mathcal{V}$ and E are finite, then $\mathcal{G}$ is a* **finite** *oriented graph. If for any $v \in \mathcal{V}$, $E(v)$ is a finite set, then $\mathcal{G}$ is called a* **locally (oriented) finite graph** *(or, simply,* **simply oriented graph***).*

For any locally oriented finite graph $\mathcal{G} = (\mathcal{V}, E, s, d)$ the set of all arcs having source v, $E = \bigcup_{v \in \mathcal{V}} E(v)$, can be denoted in the following way:

$$E(v) = \{< v, 1 >, < v, 2 >, ..., < v, k_v >\}$$

Consequently, the set of all sons of v, i.e. $S(v) = \{v_1, v_2, ..., v_{k_v}\}$, can be viewed as an ordered set. From now on we shall work only with finite ordered oriented graphs. The local ordering on the sons of a given vertex induces a total *left-to-right* order on leaves.

A **path** of length n in G from v to v' is a word $p \in E^*$, $p = e_1 e_2 ... e_n$, $n \geq 1$, where:

$$v = v_0 \stackrel{e_1}{\to} v_1 \stackrel{e_2}{\to} v_2 ... v_{n-1} \stackrel{e_n}{\to} v_n = v'$$

Note. We shall suppose that for any node v there exists a path of length 0 (from v to v) denoted by λ. We suppose the reader familiar with other basic notions concerning *connectivity, preorder traversal, depth first search, breadth search* corresponding to trees and graphs ([CLR91]).

Example 2.1.1 *Let us consider the following ordered oriented graph (tree)* $T = (\{1, 2, ..., 11\}, \{< 1,1 >, < 1,2 >, < 1,3 >, < 3,1 >, < 3,2 >, < 4,1 >, < 4,2 >, < 4,3 >, < 5,1 >, < 5,2 >\}, s, d)$, *where*

$s(< 1,1 >) = 1$ $s(< 1,2 >) = 1$ $s(< 1,3 >) = 1$ $s(< 3,1 >) = 3$
$s(< 3,2 >) = 3$ $s(< 4,1 >) = 4$ $s(< 4,2 >) = 4$ $s(< 4,3 >) = 4$
$s(< 5,1 >) = 5$ $s(< 5,2 >) = 5$ $d(< 1,1 >) = 2$ $d(< 1,2 >) = 3$
$d(< 1,3 >) = 4$ $d(< 3,1 >) = 5$ $d(< 3,2 >) = 6$ $d(< 4,1 >) = 7$
$d(< 4,2 >) = 8$ $d(< 4,3 >) = 9$ $d(< 5,1 >) = 10$ $d(< 5,2 >) = 11$

T may have the following graphical representation

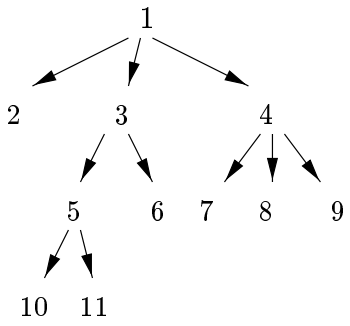


Figure 2.1.

The preorder (depth first) traversal is given by: 1,2,3,5,10,11,6,4,7,8,9.

Fundamental (needed) notions of formal languages are now presented.

Definition 2.1.4 *A* **grammar** *is a 4-tuple $G = (V_N, V_T, Z, P)$, where V_N is the alphabet of nonterminal symbols (variables), V_T is the alphabet of terminal symbols, $V = V_N \cup V_T$ is the alphabet of symbols ($V_N \cap V_T = \emptyset$), Z is the start symbol, $P \subseteq V^* \cdot V_N \cdot V^* \times V^*$ is the set of productions (or generation rules). The production (α, β) will be denoted by $\alpha \to \beta$.*

Definition 2.1.5 *(Chomsky hierarchy)*
A grammar $G = (V_N, V_T, Z, P)$ is called:

- **phrase structure** *(type 0) if no restrictions on the productions of G are imposed;*
- **context sensitive** *(type 1) if the productions are of the form $u\,x\,v \to u\,r\,v$, where $u, v \in V^*$, $x \in V_N$, $r \in V^+$ or $Z \to \lambda$ and Z does not occur in the right hand side of any production from P.*
- **context free** *(type 2) if $P \subseteq V_N \times V^*$;*
- **linear** *if $P \subseteq V_N \times (V_T^*(V_N V_T^* \cup \{\lambda\}))$;*
- **right linear** *(type 3) if $P \subseteq V_N \times V_T^*(V_N \cup \{\lambda\})$. Analogously, G is called* **left linear** *if $P \subseteq V_N \times (V_N \cup \{\lambda\})V_T^*$. Both are (called)* **regular grammars**.

For context free grammars (and (right) linear, of course), a pair $(A, \beta) \in P$ is called an $A-$production (denoted by $A \to \beta$). The productions $A \to \beta_1$, $A \to \beta_2, ...,$ $A \to \beta_k$ will be denoted sometimes by $A \to \beta_1 \,|\, \beta_2 \,|\, ... \,|\, \beta_k$. A **null production** is of the form $A \to \lambda$.

Definition 2.1.6 *Let $G = (V_N, V_T, Z, P)$ be an arbitrary grammar. We call derivation in G the binary relation (denoted by $\underset{G}{\Longrightarrow} \subseteq V^* \times V^*$) in this way:*

$(\alpha, \beta) \in \underset{G}{\Longrightarrow}$ *(i.e.* $\alpha \underset{G}{\Longrightarrow} \beta$*) iff* $\exists\ \alpha' \to \beta' \in P$ *such that* $\alpha = \alpha_1\alpha'\alpha_2$ *and* $\beta = \beta_1\beta'\beta_2$*. The transitive (and reflexive) closure of the relation* $\underset{G}{\Longrightarrow}$ *is denoted by* $\overset{+}{\underset{G}{\Longrightarrow}}$ *(*$\overset{*}{\underset{G}{\Longrightarrow}}$ *respectively).*

Definition 2.1.7 *The* **set of all sentential forms** *of the grammar* $G = (V_N, V_T, Z, P)$ *is* $SF(G) = \{\alpha \in V^* \mid \exists Z \overset{*}{\underset{G}{\Longrightarrow}} \alpha\}$. **The language generated by** G *is* $L(G) = \{w \in V_T^* \mid \exists\ Z \overset{*}{\underset{G}{\Longrightarrow}} w\}$ *(in fact,* $L(G) = SF(G) \cap V_T^*$*);*

A language L is said to be of type $j \in \{0, 1, 2, 3\}$ if there exists a grammar G of type j for which $L = L(G)$ (L is generated by G).

For context free (and linear) grammars (Definition 2.1.6) the word α' is in fact a nonterminal symbol. From now on, we shall consider only context free (or linear) grammars.

Definition 2.1.8 *A derivation is called* **left most** *(denoted* $\underset{lm}{\Longrightarrow}$*) if in every sentential form of the derivation, the* **first** *occurrence of a nonterminal symbol is replaced. Similarly, a derivation is called* **right most** *(denoted* $\underset{rm}{\Longrightarrow}$*) if in every sentential form of the derivation, the* **last** *occurrence of a nonterminal symbol is replaced.*

Definition 2.1.9 *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* $S \overset{*}{\underset{G}{\Longrightarrow}} w$*.* G *is an* **unambiguous** *grammar if it is not ambiguous.*

Definition 2.1.10 *Let* $G = (V_N, V_T, Z, P)$ *be a context free grammar.*

- $X \in V$ *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 a* **productive symbol** *if there exists a derivation* $A \overset{*}{\underset{G}{\Longrightarrow}} u$*, with* $u \in V_T^*$ *(otherwise,* A *is called* **useless***);*
- G *is a* **reduced grammar** *if all symbols from* V *are accessible and all nonterminal symbols are productive.*

Definition 2.1.11 *Let* $G = (V_N, V_T, Z, P)$ *be a context free grammar.* $A \in V_N$ *is a* **left-recursive symbol** *(***right-recursive, respectively***) if there exists a derivation* $A \overset{+}{\underset{G}{\Longrightarrow}} A\,\alpha$, $\alpha \in V^+$ *(*$A \overset{+}{\underset{G}{\Longrightarrow}} \beta\,A$, $\beta \in V^+$*).* G *is called* **left (right) recursive grammar** *if there exists a left (right) recursive symbol* $A \in V_N$*.*

Definition 2.1.12 *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* $\widetilde{P} = \{A \to \widetilde{\beta} \mid A \to \beta \in P\}$*, the* **reverse (mirror) grammar** *of* G*.*

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 (terminal or nonterminal): α, β, γ ...
- productions: $r = no(A \to \alpha)$ means that the production $A \to \alpha$ is the $r-$th one from the list of productions;
- derivations: $\overset{r}{\underset{G}{\Longrightarrow}}$ means that the production named r was applied in G; $\overset{\pi}{\underset{G}{\Longrightarrow}}$ refers to a sequence of productions π (syntactic analysis); $\overset{0}{\underset{G}{\Longrightarrow}}$ means that no production has been applied, i.e. $\alpha \overset{0}{\underset{G}{\Longrightarrow}} \alpha$;
- let $\alpha = \alpha_1 \, \alpha_2 \, ... \, \alpha_k$ be a word over V. Then
  - ${}^{(m)}\alpha = \begin{cases} \alpha_1 \, \alpha_2 \, ... \, \alpha_m & \text{if } m \leq k \\ \alpha & \text{otherwise} \end{cases}$
  - $\alpha^{(n)} = \begin{cases} \alpha_{k-n+1} \, \alpha_{k-n+2} \, ... \, \alpha_k & \text{if } n \leq k \\ \alpha & \text{otherwise} \end{cases}$
- $\mathbf{N}$ denotes the set of natural numbers, $\mathbf{N}^+$ denotes the set of strict positive natural numbers.

Example 2.1.2 *Let us consider the linear grammar* $G = (\{Z\}, \{a, b\}, Z, \{Z \to a\,b \mid a\,Z\,b\})$. *The language generated by* G *is* $L(G) = \{a^n\,b^n \mid n \geq 1\}$. *This can be immediately prove by showing (for instance, by induction on* k*) that*

$$\{\alpha \mid \alpha \in \{Z, a, b\}^*, \exists Z \overset{k,*}{\underset{G}{\Longrightarrow}} \alpha\} = \{a^k\,Z\,b^k, a^k\,b^k\}$$

2.1.2 Some properties of context free and linear languages

Definition 2.1.13 *A* **derivation tree** $T = (\mathcal{V}, E, s, d)$ *in a context free grammar* $G = (V_N, V_T, Z, P)$ *is a node labeled finite ordered oriented tree. The labels of the nodes are given by a function* $f : \mathcal{V} \to V_N \cup V_T \cup \{\lambda\}$.

For any $v \in \mathcal{V}$, *with* $S(v) = \{v_1, v_2, ..., v_k\}$, *if* $f(v) = X$, $f(v_1) = Y_1$, $f(v_2) = Y_2$, ..., $f(v_k) = Y_k$, *then* G *contains the production* $X \to Y_1Y_2...Y_k$.

If $f(v) = X$ *(*v *being the root) and the word* $w = v_1v_2...v_n$ *is formed by the (ordered) labels of the leaves, we say that* T **describes the word** w **generated from** X. *If* $X = Z$ *and* $w \in V_T^*$, *then the corresponding derivation tree* T *describes a word of* $L(G)$.

Theorem 2.1.1 *(Locality property for context free languages, [HoU79], [Sal73])*
Let $G = (V_N, V_T, Z, P)$ be a context free grammar and let $Y_1...Y_n \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ be an arbitrary derivation. Then there exists $\alpha_1 \in V^*$, ..., $\alpha_n \in V^*$ so that $\alpha = \alpha_1...\alpha_n$ and $Y_i \overset{*}{\underset{G}{\Longrightarrow}} \alpha_i$, $\forall\ i = \overline{1,n}$.

Theorem 2.1.2 *(Pumping Lemma for Linear Languages, [Sal73])*
For every linear language $L \subseteq V_T^*$, there exists a natural number N, depending only on L, such that if $z \in L$ with $|z| > N$ then there exist $u, v, w, x, y \in V_T^*$ for which the following conditions are fulfilled:

(a) $z = u\,v\,w\,x\,y$;

(b) $|v\,x| > 0$;

(c) $|u\,v\,x\,y| \leq N$;

(d) $\forall\ i \geq 0 :\ u\,v^i\,w\,x^i\,y \in L$.

In the following, we give the definitions of some most important subclasses of context free languages, for which the membership problem can be deterministically solved in linear time on the length of the input word.

The class of $TD(k)$ grammars has been introduced in 1968 by P. M. Lewis and R. E. Stearns ([LeS68]). Later, this was recalled $LL(k)$. The name $TD(k)$ comes from **T**op-**D**own analysis of the input word using **k** lookahead symbols and the name $LL(k)$ comes from **L**eft to right scanning of the input constructing a **L**eftmost derivation, using **k** lookahead symbols.

Definition 2.1.14 *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:*

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

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

$$^{(k)}v_1 =^{(k)} v_2$$

then $\beta_1 = \beta_2$.

In 1965, D. E. Knuth introduced the class of $LR(k)$ grammars ([Knu65]). The name $LR(k)$ comes from: **L**eft to right scanning of the input constructing a **R**ightmost derivation, using **k** lookahead symbols.

Definition 2.1.15 *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:*

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

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

$$^{(k)}u =^{(k)} v$$

then $\alpha = \alpha'$, $A = A'$ and $\beta = \beta'$.

The precedence grammars were invented in 1963 by R. W. Floyd ([Flo63]). Next, N. Wirth described in 1965 an algorithm for "finding" the precedence functions ([Wir65]). Later, in 1966, N. Wirth and H. Weber have provided (using precedence grammars) a formal definition for the language Euler - a generalization of ALGOL ([WiW66]). In 1968, N. Wirth gave a grammar for PL360 ([Wir68]).

Definition 2.1.16 *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 $\cdot> \subseteq V \times V_T$:*

- *$X <\cdot Y$ 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 $A \to \alpha\, X\, Y\, \beta \in P$;*
- *$X \cdot> a$ 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 2.1.17 *A context free grammar G without null productions in which the binary relations $<\cdot$, $\doteq$, $\cdot>$ are disjoint, is called a* **precedence grammar**. *G is called* **invertible grammar** *if the statement holds:*

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

G is called a **simple precedence grammar** *(SP grammar) if it is a precedence and* **invertible grammar**.

The above defined subclasses of grammars may be combined using a *mirroring process* such that bidirectional parsers may be derived for recognizing an input word (Chapters 4 and 5).

2.2 Attribute grammars

We present a unified theoretical approach for attribute grammars. The circularity problem for these grammars is treated in detail ([Knu68], [Alb91a], [And97],[ASU86]).

Attribute grammars have proved to be a useful formalism for specifying the context sensitive syntax and the semantics of programming languages, as well as for implementing editors, compilers and compiler-writing systems.

Attribute grammars are extensions of context free grammars in the sense that the information associated with programming languages constructs may be attached to grammar symbols representing these constructs, named *attributes*. Each attribute has a (possibly infinite) set of possible values. Attribute values are defined by attribute evaluation rules associated with the productions of the context free grammar. These rules specify how to compute the values of certain attribute occurrences as a function of other attribute occurrences.

The attributes associated with a grammar symbol are divided into two disjoint classes, the *synthesized attributes* and the *inherited attributes*. The attribute evaluation rules associated with a grammar production define the synthesized attributes attached to the grammar symbol on the left hand side and the inherited attributes attached to the grammar symbols on the right hand side of the production.

In a context free grammar, a tree structure to each sentence may be assigned. One could think of the nodes (grammar symbols) in a parse tree as records with fields for holding information, whose names correspond to attributes. The values of the synthesized attributes in a parse tree node and the inherited attributes in its immediate descendants are defined by the attribute evaluation rules associated with the production applied for that node. The value of a synthesized attribute of a parent is computed from the values of the attributes of its children and (possibly) other attributes of the parent itself. The values of an inherited attribute of a child are computed from the values of the attributes of its parent and its siblings and (possibly) attributes of the child itself.

Generally speaking, a synthesized attribute attached to a tree node contains information concerning the subtree originating at that node. Inherited attributes are convenient for expressing the dependence of a programming language construct of the context in which it occurs.

Several (different) definitions of attribute grammars are now used. In Section 2.2.1, we give only one of them which we think that it is the most appropriate for describing in a formal way the link with programming languages.

2.2.1 Definitions, notations, examples

Definition 2.2.1 *An* **attribute grammar** *([Alb91a]) is a five-tuple*

$$AG = (G, SD, AD, R, C),$$

defined as follows:

(1) $G = (V_N, V_T, Z, P)$ *is a (the underlying) context free grammar (G is assumed to be reduced).*

- *(1.1)* V_N *and* V_T *denote the alphabets of nonterminal and, respectively, terminal symbols, and form the vocabulary* $V = V_N \cup V_T$, $V_N \cap V_T = \emptyset$;
- *(1.2)* P *is the finite set of productions; a production* $p \in P$ *will be denoted as* $p : X_{p0} \rightarrow X_{p1} \ldots X_{pn_p}$, *where* $n_p \geq 0$, $X_{p0} \in V_N$ *and* $X_{pk} \in V$ *for* $1 \leq k \leq n_p$;
- *(1.3)* $Z \in V_N$ *is the start symbol, which does not occur on the right hand side of any production.*

(2) $SD = (TYPE - SET, FUNC - SET)$ *is a semantic domain.*

- *(2.1)* $TYPE - SET$ *is a finite set of sets;*

(2.2) $FUNC - SET$ *is a finite set of total functions of type* $type_1 \times ... \times type_n \rightarrow type_0$, *where* $n \geq 0$ *and* $type_i \in TYPE - SET$ $(0 \leq i \leq n)$.

(3) $AD = (A, I, S, TYPE)$ *is a description of attributes.*

(3.1) For each symbol $X \in V$ *there exists a set* $A(X)$ *of attributes which can be partitioned into two disjoint subsets* $I(X)$ *and* $S(X)$ *of* **inherited** *and* **synthesized** *attributes, respectively;*

(3.2) The set of all attributes will be denoted by A, *i.e.* $A = \bigcup_{X \in V} A(X)$.

(3.3) Attributes associated with different symbols are considered as different, i.e. $A(X) \cap A(Y) = \emptyset$ *if* $X \neq Y$. *If necessary, an attribute* `a` *of symbol* X *will be denoted by* $X.a$;

(3.4) For `a`$\in A$, $TYPE(a) \in TYPE - SET$ *is the set of possible values of* `a`.

(4) $R(p)$ *is a finite set of attribute evaluation rules (semantic rules) associated with the production* $p \in P$.

(4.1) Production $p : X_{p0} \rightarrow X_{p1} ... X_{pn_p}$ *is said to have the* **attribute occurrence** (a, p, k) *if* $a \in A(X_{pk})$;

(4.2) The set of all attribute occurrences of production p *will be denoted by* $AO(p)$;

(4.3) The set $AO(p)$ *can be partitioned into two disjoint subsets of* **defined occurrences** *and* **used occurrences** *denoted by* $DO(p)$ *and* $UO(p)$, *respectively:*

$$DO(p) = \{(s, p, 0) \mid s \in S(X_{p0})\} \cup \{(i, p, k) \mid i \in I(X_{pk}) \wedge 1 \leq k \leq n_p\}$$

$$UO(p) = \{(i, p, 0) \mid s \in I(X_{p0})\} \cup \{(s, p, k) \mid i \in S(X_{pk}) \wedge 1 \leq k \leq n_p\}$$

The attribute evaluation rules of $R(p)$ *specify how to compute the values of the attribute occurrences in* $DO(p)$ *as a function depending on the values of certain other attribute occurrences in* $AO(p)$. *The evaluation rule defining the attribute occurrence* (a, p, k) *has the form*

$$(a, p, k) := f((a_1, p, k_1), ..., (a_m, p, k_m))$$

$(a, p, k) \in DO(p)$, $f : TYPE(a_1) \times ... \times TYPE(a_m) \rightarrow TYPE(a)$, $f \in FUNC - SET$ *and* $(a_i, p, k_i) \in AO(p)$ *for* $1 \leq k \leq m$. *We say that* (a, p, k) **depends on** (a_i, p, k_i), *for* $1 \leq i \leq m$.

(5) $C(p)$ *is a finite set of semantic conditions associated with the production* p. *These conditions are predicates of the form*

$$\pi((a_1, p, k_1), ..., (a_m, p, k_m))$$

$\pi : TYPE(a_1) \times ... \times TYPE(a_m) \rightarrow \{true, false\}$, $\pi \in FUNC - SET$, *and* $(a_i, p, k_i) \in AO(p)$ *for* $1 \leq i \leq m$.

Semantic conditions allow the specification of a subset of the language defined by the underlying context free grammar. A sentence that is generated by G is a sentence of the language specified by AG if the semantic conditions yield true. Traditionally, the definitions of attribute grammars require that both the start symbol and the terminal symbols to have no inherited attributes. We do not assume this restriction.

We have been so far concerned with the syntax of attribute grammars. Let us describe their semantics.

An unambiguous context free grammar assigns a single derivation tree to each of its sentences. The nodes of a derivation tree are labeled with symbols from V. For each interior node there exists a production $X_{p0} \to X_{p1} \ldots X_{pn_p}$, such that the node is labeled with X_{p0} and its n_p sons are labeled with X_{p1}, ..., X_{pn_p}, respectively. We say that p is the production *(applied)* at that node.

Definition 2.2.2 *A derivation tree is* **complete** *if it has only terminal symbols (or the empty string) as labels of its leaves and the start symbol as the label of its root.*

Unless otherwise stated our derivation trees will be assumed to be complete.

Definition 2.2.3 *Given a derivation tree in an attribute grammar* $AG = (G, SD, AD, R, C)$, *instances of attributes are attached to the nodes in the following way: if node* N *is labeled with grammar symbol* X, *then for each attribute* "a" $\in A(X)$ *an instance of* "a" *is attached to node* N. *We say that the derivation tree has the attribute instance* $N.a$. *Let* N_0 *be a node,* p *a production at* N_0 *and* N_1, ..., N_{n_p} *the sons of* N_0 *in the given order (Definition 2.1.13). An* **attribute evaluation instruction**

$$N_k.a := f(N_{k_1}.a_1,\ \ldots,\ N_{k_m}.a_m)$$

is associated with attribute instance $N_k.a$ *if the attribute evaluation rule*

$$(a, p, k) := f((a_1, p, k_1),\ \ldots,\ (a_m, p, k_m))$$

is associated with production p. *We say that the attribute instance* $N_k.a$ depends *on the attribute instance* $N_{k_i}.a_i$ *for* $1 \leq i \leq m$. *If all the values are known and satisfy all attribute evaluation rules then we say that the attributed derivation tree is* **consistent.**

Definition 2.2.4 *A* **decorated** *(or* **attributed***)* **derivation tree** *is a derivation tree in which all attribute instances have a value (which is not necessarily consistent). A* **consistently decorated** *(***attributed***)* **derivation tree** *is a derivation tree in which all attribute instances are defined according to their associated attribute evaluation instructions, i.e. the "execution" of any evaluation instruction does not change the values of the attribute associated with a tree node (as described in Definition 2.2.3).*

In this way, in an attribute grammar a (consistently) decorated derivation tree may be assigned to each of its sentences.

Definition 2.2.5 *For each derivation tree T a* **dependency graph** *$D(T)$ can be defined by taking the attribute instances of T as its vertices. The directed arc $(N_i.a, N_j.b)$ is contained in the graph if and only if the attribute instance $N_j.b$ depends on the attribute instance $N_i.a$. A path in a dependency graph will be called a* **dependency path.** *For $n > 0$, $dp[N_1.a_1, N_2.a_2, ..., N_n.a_n]$ stands for a path with arcs $(N_1.a_1, N_2.a_2)$, $(N_2.a_2, N_3.a_3)$, ..., $(N_{n-1}.a_{n-1}, N_n.a_n)$. A path $dp[N_1.a_1, N_2.a_2, ..., N_n.a_n, N_1.a_1]$ will be called a* **circular** *dependency path. An attribute grammar is* **circular** *if it has a derivation tree whose dependency graph contains a circular dependency graph. An attribute grammar is called* **non-circular (well defined)** *if it is not circular. The class of all well defined grammars is denoted by WAG.*

The task of an *attribute evaluator* is to compute the values of all attribute instances attached to a derivation tree, by executing the attribute evaluation instructions associated with these attribute instances. Generally, the order of the evaluation is not important. The only restriction may be that an attribute evaluation instruction cannot be executed before its arguments are available. An attribute instance is *available* if its value is defined, otherwise it is *unavailable*. At the beginning, all attribute instances attached to a derivation tree are unavailable, with the exception of the inherited attribute instances attached to the root (containing information concerning the environment) and the synthesized attribute instances attached to the leaves (determined by the parser). At each step an attribute instance whose value can be computed is chosen. The evaluation process continues until all attribute instances in the tree are defined or until none of the remaining attribute instances can be evaluated.

For a traditional attribute evaluator (as described above) it is impossible to evaluate attribute instances involved in a circular dependency path.

Example 2.2.1 *Let $AG_1 = (G_1, SD_1, AD_1, R_1, C_1)$ be the following attribute grammar:*

(1) $G_1 = (\{Z, A\}, \{a, b\}, P_1, Z_1)$ the underlying context free grammar and P_1 given below;

(2) $SD_1 = (\{integer\}, FUNC - SET_1)$, where $FUNC - SET_1$ is described below (it contains the identity function, constant function, add function, etc.);

(3) $AD_1 = (A_1, I_1, S_1, TYPE_1)$, where

(3.1) $A_1 = \{i, s\}$;

(3.2) $I_1(Z) = I_1(A) = \{i\}$;

(3.3) $S_1(Z) = S_1(A) = \{s\}$;

(3.4) $TYPE_1(i) = TYPE_1(s) = \{integer\}$;

(4) the set R_1 of attribute evaluation rules is described below;

(5) the set C_1 is also presented below together with P_1 and R_1.

Because a production (e.g. second production of G_1) might contain an occurrence of the same nonterminal symbol X in the attribute evaluation rule, X will have an index (starting from 1 to the last occurrence). In the following, the sets P_1, R_1 and C_1 are presented.

Production 1**:**
$Z \to A$
Attribute evaluation rules:
$Z.i := 1$; $Z.s := A.s$ $A.i := Z.i$

Production 2**:**
$A \to a\,A$
Attribute evaluation rules:
$A_2.i := A_1.i + 1$; $A_1.s := A_2.s + 1$;

Production 3**:**
$A \to b$
Attribute evaluation rule:
if $A.i > 10$ **then** $A.s := 0$ **else** $A.s := 1$

Let us consider the word $w = aab$. Figure 2.2 pictures the corresponding derivation tree T and the dependency graph $D(T)$.

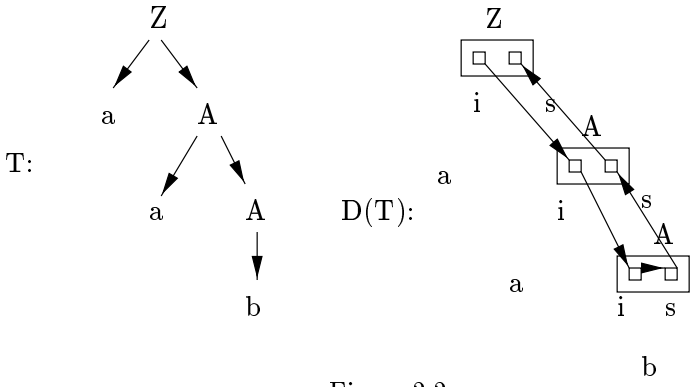


Figure 2.2.

In the following example ([ASU86]) we construct an attribute grammar for "keeping track" the moves of the robot (and providing the final position). Suppose that a robot can be instructed to move one step *east*, *north*, *west*, or *south* from its current position.

Example 2.2.2 *Let $AG_2 = (G_2, SD_2, AD_2, R_2, C_2)$ be the following attribute grammar:*

(1) $G_2 = (\{seq, instr\}, \{begin, east, north, west, south\}, P_2, seq)$, where the productions P_2 will be given below;

(2) $SD_2 = (\{integer\}, FUNC - SET_2)$, *where the set of total functions* $FUNC - SET_2$ *are presented below (the constant functions, sum etc.);*

(3) $AD_2 = (\{x, y, dx, dy\}, \emptyset, S_2, TYPE_2)$, *where:*

$S_2(seq) = \{x, y\}$;
$S_2(instr) = \{dx, dy\}$;
$TYPE_2(x) = TYPE_2(y) = TYPE_2(dx) = TYPE_2(dy) = integer$
(we may write $TYPE_2(dx) = TYPE_2(dy) = \{-1, 0, 1\}$*);*

(4) R_2 *is described below together with the productions* P_2*;*

(5) $C_2 = \emptyset$.

In the following, the sets P_2 *and* R_2 *are presented.*

Production 1:
$seq \to begin$
Attribute evaluation rules:
$seq.x := 0$;
$seq.y := 0$

Production 2:
$seq_1 \to seq_2\ instr$
Attribute evaluation rules:
$seq_1.x := seq_2.x + instr.dx$;
$seq_1.y := seq_2.y + instr.dy$

Production 3:
$instr \to east$
Attribute evaluation rules:
$instr.dx := 1$;
$instr.dy := 0$

Production 4:
$instr \to north$
Attribute evaluation rules:
$instr.dx := 0$;
$instr.dy := 1$

Production 5:
$instr \to west$
Attribute evaluation rules:
$instr.dx := -1$;
$instr.dy := 0$

Production 6:
$instr \to south$
Attribute evaluation rules:
$instr.dx := 0$;
$instr.dy := -1$

The abbreviations for the attributes, as the pair (x, y), *mean that* x *and* y *are the number of steps to the east and north, respectively, from the starting position (if* x *is negative, then the robot is to the west of the starting position; similarly, if* y *is negative, then the robot is to the south of the starting position).*

In the following example ([And97]), we present a more natural attribute grammar using both synthesized and inherited attributes which describe the same language.

Example 2.2.3 *Let* $AG_3 = (G_3, SD_3, AD_3, R_3, C_3)$ *be the following attribute grammar:*

(1) $G_3 = (\{Z, seq\}, \{east, north, west, south\}, P_3, Z)$, *where the productions* P_3 *will be given below;*

(2) $SD_3 = (\{integer\}, FUNC - SET_3)$, *where the set of total functions* $FUNC - SET_3$ *are presented below (the constant functions, sum etc.);*

(3) $AD_3 = (\{x, y, newx, newy\}, I_3, S_3, TYPE_3)$, *where:*

$S_3(Z) = \{newx, newy\}$;

$S_3(seq) = \{newx, newy\}$;

$I_3(seq) = \{x, y\}$;

$TYPE_3(x) = TYPE_3(y) = TYPE_3(newx) = TYPE_3(newy) = integer$;

(4) R_3 *is described below together with the productions* P_3;

(5) $C_3 = \emptyset$.

In the following, the sets P_3 *and* R_3 *are presented.*

Production 1:
$Z \rightarrow seq$
Attribute evaluation rules:
$Z.newx := seq.newx$;
$Z.newy := seq.newy$;
$seq.x := 0$;
$seq.y := 0$

Production 2:
$seq_1 \rightarrow east\ seq_2$
Attribute evaluation rules:
$seq_1.newx := seq_2.newx$;
$seq_1.newy := seq_2.newy$;
$seq_2.x := seq_1.x + 1$;
$seq_2.y := seq_1.y$

Production 3:
$seq_1 \rightarrow north\ seq_2$
Attribute evaluation rules:
$seq_1.newx := seq_2.newx$;
$seq_1.newy := seq_2.newy$;
$seq_2.x := seq_1.x$;
$seq_2.y := seq_1.y + 1$

Production 4:
$seq_1 \rightarrow west\ seq_2$
Attribute evaluation rules:
$seq_1.newx := seq_2.newx$;
$seq_1.newy := seq_2.newy$;
$seq_2.x := seq_1.x - 1$;
$seq_2.y := seq_1.y$

Production 5:
$seq_1 \rightarrow south\ seq_2$
Attribute evaluation rules:
$seq_1.newx := seq_2.newx$;
$seq_1.newy := seq_2.newy$;
$seq_2.x := seq_1.x$;
$seq_2.y := seq_1.y - 1$

Production 6:
$seq \rightarrow \lambda$
Attribute evaluation rules:
$seq.newx := seq.x$;
$seq.newy := seq.y$

In [Knu68] an important subclass of attribute grammars (the so called *purely synthesized AG's* (SAG)) has been defined. In the same paper, it was proved that SAG's have the same recognizing power as Turing machines. We can conclude that the power of attribute grammars is the same as that of the Turing machines.

2.2.2 The circularity problem

For any derivation tree T, we can define (Definition 2.2.5) a dependency graph $D(T)$ whose arcs specify the dependency relations. These relations express the fact that certain attribute instances must be computed before others. Clearly, the attribute evaluation rules are well defined if and only if no dependency graph contains an oriented cycle.

Definition 2.2.6 *An attribute grammar is said to be* **noncircular (well defined)** *if for any derivation tree T the dependency graph $D(T)$ contains no oriented cycle.*

For any dependency graph $D(T)$ of a noncircular attribute grammar AG a "topological sorting algorithm" can be applied to produce a linear list

$$(N_1.a_1, N_2.a_2, ..., N_n.a_n)$$

of its vertices. The sequence of vertices in the list is such that for any arc $(N_i.a_i, N_j.a_j)$ in $D(T)$ we have $i < j$. This implies that a vertex $N_j.a_j$ can never precede a vertex $N_i.a_i$ in the list if $N_j.a_j$ depends on $N_i.a_i$. I.e., if the attribute instances are evaluated in the order of the occurrences of their associated vertices in the list, then the required arguments of their evaluation instructions have already been evaluated.

Up to now, we have considered the dependencies between attribute instances in complete derivation trees. From now on we shall be also interested in dependencies between attribute instances in subtrees of complete derivation trees. A subtree of a complete derivation tree has (as usual) only terminal symbols as labels of its leaves, but is allowed to have any symbol of V (not only the start symbol Z) as the label of its root. We can define a dependency graph $D(T)$ for a subtree T in a similar way to the definition of a complete derivation tree.

Definition 2.2.7 *Let $p: \ X_{p0} \to X_{p1}...X_{pn_p}$ be an arbitrary production in G. We associate a* **dependency graph** *DG_p as follows:*

- *the vertices of DG_p are the attribute occurrences of the production p (i.e. the attribute occurrences of $X_{p0}, X_{p1}, ..., X_{pn_p}$);*
- *for every pair of the attribute occurrences (a, p, j) and (b, p, k) of the production p there exists a (directed) arc from (a, p, j) to (b, p, k) in DG_p if and only if (b, p, k) depends on (a, p, j).*

If T has a terminal symbol as the label of its root, $D(T)$ may have vertices, but no arcs. If the root of T is labeled with a nonterminal symbol, then T has the form:

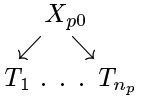


Figure 2.3

for some production $p : X_{p0} \to X_{p1}...X_{pn_p}$ applied to the root of T, where T_j is a derivation tree with X_{pj} labeling its root, for $1 \leq j \leq n_p$. $D(T)$ is obtained from DG_p, $D(T_1)$, ..., $D(T_{n_p})$ by identifying the vertices for the attribute occurrences of X_{pj} in DG_p with the corresponding vertices for the attribute instances attached to the root of T_j in $D(T_j)$ $(1 \leq j \leq n_p)$.

We now concentrate on the problem of finding oriented cycles in dependency graphs, i.e. to establish the (non)circularity of an attribute grammar (Definition 2.2.6). Consider a complete derivation tree T which includes an oriented cycle. Let T_0 be the smallest subtree of T containing this cycle and let T_0 be constructed from production p and subtrees T_1, ..., T_{n_p} (Figure 2.3). The fact that T_0 is the smallest subtree which includes the cycle implies that the cycle runs through DG_p. With respect to the subtrees T_j $(1 \leq j \leq n_p)$ we are not interested in the details of their dependency paths, but just in the fact that a dependency path runs from an inherited attribute instance of the root to a synthesized instance of the root.

Notation 2.2.1 *Let T be a complete derivation tree. A subtree of T with root N will be denoted by T/N.*

Definition 2.2.8 *Let T be a complete derivation tree. With each subtree T/N we can associate a directed graph $IS(T/N)$, representing its $i-$**to**$-s$ **behavior** at N as follows:*

- *the vertices of $IS(T/N)$ are the attributes of X, where X is the label of N;*
- *for each pair of inherited and synthesized attributes $X.i$, $X.s$, respectively, an arc from $X.i$ to $X.s$ is included in $IS(T/N)$ if and only if there is an oriented path between the corresponding vertices $N.i$ and $N.s$ in $D(T/N)$.*

We do not restrict our computations to a single derivation tree and a single node of this tree (we consider any node of any possible derivation tree). For every $X \in V$ we need to compute the dependency graphs (with vertices from $A(X)$). These show how the synthesized attributes of X depend on the inherited attributes of X (denoted by $i-$to$-s$ attribute dependency for subtrees whose root is labeled X). This set of graphs will be called $IS-SET(X)$ and is defined as follows.

Definition 2.2.9 *For each $X \in V$, $IS-SET(X) = \{IS(T/N) \mid T$ is a derivation tree and N is a node of T labeled $X\}$.*

Note that for each $X \in V$, $IS - SET(X)$ is finite since $A(X)$ is finite. The elements of $IS - SET(X)$ will be called the $is-$graphs of X.

In a similar way we can define graphs "showing" how the inherited attributes associated with a grammar symbol X may depend on the synthesized attributes.

Notation 2.2.2 *Consider a derivation tree T whose subtree with root N has been deleted, excluding N itself. Such an (incomplete) derivation tree will be denoted by $T - T/N$.*

Definition 2.2.10 *Let T be a complete derivation tree. With each tree $T - T/N$ we can associate a directed graph $SI(T - T/N)$, representing the $s-$**to**$-i$* **behavior** *at N as follows:*

- *the vertices of $SI(T - T/N)$ are the attributes of X, where X is the label of N;*
- *for each pair of synthesized and inherited attributes $X.s$, $X.i$, respectively, an arc from $X.s$ to $X.i$ is included in $SI(T - T/N)$ if and only if there exists an oriented path between $N.s$ and $N.i$ in $D(T - T/N)$.*

The set of graphs, expressing the different $s-$to$-i$ attribute dependency patterns at X, of trees whose subtree with root labeled X has been deleted, will be called $SI - SET(X)$ and is defined as follows.

Definition 2.2.11 *For each $X \in V$, $SI - SET(X) = \{SI(T - T/N) \mid T$ is a derivation tree and N is a node of T labeled $X\}$.*

The elements of $SI - SET(X)$ will be called the $si-$graphs of X.

For each production p we have already defined a dependency graph DG_p. The following notation is needed to express the combination between the dependency information of a production and its context.

Notation 2.2.3 *Let p be a production $p : X_0 \rightarrow X_{p1}...X_{pn_p}$ and let D_i, for $0 \leq i \leq n_p$, be a directed graph with vertices $A(X_{pi})$. Then $DG_p[D_0, D_1, ..., D_{n_p}]$ is the directed graph obtained from DG_p by adding an arc from the attribute occurrence (a, p, i) to (b, p, i) whenever there exists an arc from the attribute $X_{p_i}.a$ to the attribute $X_{p_i}.b$ in D_i $(0 \leq i \leq n_p)$.*

For the case where D_0 is an $si-$graph from $SI - SET(X_{p0})$ and D_i $(1 \leq i \leq n_p)$ an $is-$graph from $IS - SET(X_{pi})$ $(1 \leq n \leq n_p)$, $DG_p[D_0, D_1, ..., D_{n_p}]$ describes the (indirect) dependencies between attribute instances of an application of production p within a derivation tree.

To compute the set $IS - SET(X)$, we isolate the subtrees with root N from the surrounding trees, and to compute the set $SI - SET(X)$ we consider the incomplete derivation trees from which a subtree with root N has been deleted. Hence, we need to consider the case of a "hole" in the context of a production.

Notation 2.2.4 *Given a production $p : X_{p0} \rightarrow X_{p1}...X_{pn_p}$ and the directed graphs D_i $(0 \leq i \leq n_p)$ with vertices $A(X_{pi})$, then*

$$DG_p - k[D_0, D_1, ..., D_{k-1}, D_{k+1}, ..., D_{n_p}] = DG_p[D_0, D_1, ..., D_{n_p}],\ 0 \leq k \leq n_p$$

To directly express the indirect dependencies directly we denote $DG_p - k[...]^*$, that is the transitive closure of $DG_p - k[...]$ (for simplicity, the explicit arguments of $DG_p - k$ have been indicated by dots). From $DG_p - k[...]^*$, a new directed graph $DG_p - k^*[...]$ can be constructed as follows.

Definition 2.2.12 *The vertices of* $DG_p - k^*[...]$ *are the attributes of* $A(X_{pk})$. *For each pair of attributes* $X_{pk}.a$, $X_{pk.b}$, *an arc from* $X_{pk}.a$ *to* $X_{pk.b}$ *is included in* $DG_p - k^*[...]$ *if and only if there exists an arc from* (a, p, k) *to* (b, p, k) *in* $DG_p - k[...]^*$.

$DG_p - 0^*[...]$ can be used to characterize the $is-$graphs as follows.

Lemma 2.2.1 *For each* $X \in V_T$, $IS - SET(X)$ *represents the set consisting of the single graph with vertices* $A(X)$ *and no arcs. For each* $X \in V_N$,

$$IS - SET(X) = \{DG_p - 0^*[D_1, ..., D_{n_p}] \mid p \in P,\ X_{p0} = X,$$

$$D_i \in IS - SET(X_{pi})\ (1 \leq i \leq n_p)\}.$$

In fact, $IS - SET(X)$ *are the smallest sets of graphs satisfying the above equations.*

Proof Following [And97]. ■

From Lemma 2.2.1 (the recursive characterization of $IS-SET(X)$) the following algorithm ([Knu68],[Alb91a],[And97]) can be derived.

Algorithm 3.1. Computation of the sets $IS - SET(X)$

Input: An arbitrary attribute grammar AG;
Output: The sets $IS - SET(X)$ for all $X \in V$.
Method:
begin
  for (all $X \in V_N$) **do** $IS - SET(X) := \emptyset$;
  for (all $X \in V_T$) **do**
    $IS - SET(X)$:= the set consisting of the single graph with vertices $A(X)$ and no arcs;
  repeat {for all $X \in V_N$: add further graphs to the set $IS - SET(X)$}
    choose a production $p: X_{p0} \to X_{p1}...X_{pn_p}$ for which
    none of the sets $IS - SET(X_{pi})$ $(1 \leq i \leq n)$ is empty;
    for $(1 \leq i \leq n_p)$ **do** choose a graph D_i in $IS - SET(X_{pi})$;
    if (graph $DG_p - 0^*[D_1, ..., D_{n_p}]$ is not in $IS - SET(X_{p0})$ **then**
      add this graph to $IS - SET(X_{p0})$
  until no further graphs can be added to any set $IS - SET(X)$
end.

Note that for all $X \in V$, $IS - SET(X)$ is non-empty. This follows immediately from the fact that the underlying context free grammar of AG is reduced.

We can also formulate a related, recursive, characterization for the sets $SI - SET(X)$, based on the graphs $DG_p - k^*[...]$ $(k \neq 0)$.

Lemma 2.2.2 $SI - SET(Z)$ *represents the set consisting of the single graph with vertices* $A(Z)$ *and no arcs. For each* $X \in V$ *such that* $X \neq Z$,

$$SI-SET(X) = \{DG_p-k^*[D_0, D_1, ..., D_{k-1}, D_{k+1}, ..., D_{n_p}] \mid p \in P,\ 1 \leq k \leq n_p,$$

$X_{pk} = X,\ D_0 \in SI{-}SET(X_{p0}),\ D_i \in IS{-}SET(X_{pi})\ (1 \leq i \leq n_p,\ i \neq k)\}$.

In fact, the $SI - SET(X)$ are the smallest sets of graphs satisfying the above equations.

Proof In [And97]. ■

This characterization leads to the following algorithm ([Alb91a, And97]).

Algorithm 3.2. Computation of the sets $SI - SET(X)$

Input: An arbitrary attribute grammar AG and the sets $IS - SET(X)$ for all $X \in V$;
Output: The sets $SI - SET(X)$ for all $X \in V$.
Method:
begin
$SI - SET(Z)$:= the set consisting of the single graph with vertices $A(Z)$ and no arcs;
for (all $X \in V,\ X \neq Z$) **do** $SI - SET(X) := \emptyset$;
repeat {for all $X \in V$: add further graphs to the set $SI - SET(X)$}
choose a production $p:\ X_{p0} \to X_{p1}...X_{pn_p}$
choose a graph D_0 in $SI - SET(X_{p0})$
choose an integer k $(1 \leq k \leq n_p)$
for $(1 \leq i \leq n_p)$ **do** choose a graph D_i in $IS - SET(X_{pi})$;
if (graph $DG_p - k^*[D_0, D_1, ...D_{k-1}, D_{k+1}, ..., D_{n_p}] \notin SI - SET(X_{pk})$)
then add this graph to $SI - SET(X_{pk})$
until no further graphs can be added to any set $SI - SET(X_{pk})$
end.

Having the sets $IS - SET(X)$ for all $X \in V$, the circularity test is straightforward.

Algorithm 3.3. Circularity test

Input: An arbitrary attribute grammar AG and the sets $IS - SET(X)$ for all $X \in V$;
Output: The graphs $DG_p - 0[...]$ containing an oriented cycle.
Method:
begin
for (every production $p: X_{p0} \to X_{p1}...X_{pn_p}$) **do**
for (each combination $[D_1, ..., D_{n_p}]$, where $D_i \in IS - SET(X_{pi})$
$(1 \leq i \leq n_p)$) **do**
if (graph $DG_p - 0[D_1, D_2, ..., D_{n_p}]$ contains an oriented cycle)
then output this graph
end.

Theorem 2.2.1 *The attribute grammar AG is noncircular if and only if $DG_p - 0[D_1, ..., D_{n_p}]$ contains no oriented cycle, for any $p : X_{p0} \to X_{p1}...X_{pn_p} \in P,\ D_i \in IS - SET(X_{pi})\ (1 \leq i \leq n_p)$.*

Proof In [And97]. ■

Algorithms 3.1 and 3.3 form together Knuth's algorithm for testing the circularity of attribute grammars ([Knu71]). M. Jazayeri, W. F. Ogden and W. C. Rounds ([JOR75a], [JOR75b], [Jaz81]) showed that the time complexity of the circularity test is inherently exponential. However, the only exponential factor in the complexity of the algorithm is the number of graphs in the set $IS - SET(X)$ constructed for each nonterminal symbol. K. L. Räihä and M. Saarinen [RaS77] found that for "practical" grammars this number is very small (at most 4) and discussed techniques to improve the implementation of the algorithm to compute the sets $IS - SET(X)$. Their experiments showed that for practical grammars the computation of the dependencies is feasible. Implementation aspects of the circularity test are also considered by B. Lorho and C. Pair [LoP85], K. S. Chebotar [Che81], P. Deransart, M. Jourdan and B. Lorho [DJL84].

Much more, there exists an hierarchy of important subclasses of attribute grammars.

$$WAG \supset ANCAG \supset PAG \supset OAG \supset LAG \supset SAG,$$

where the suffix "AG" comes from *attribute grammars*, and the others from "W" - *well defined*, "ANC" - *absolutely non-circular*, "PAG" - *partitioned*, "O" - *ordered*, "L" - *left* and "S" - *synthesized*.

To evaluate the attribute values in the subclasses WAG [Knu68], ANCAG [KeW76] (sometimes called SNCAG = strongly non-circular attribute grammars, [Jou84], [JPJ90]), PAG [WaG84], OAG [Kas80] the structure tree must be kept in memory. This is not the case for the subclasses LAG, SAG [LRS74] and consequently there exists polynomial algorithms for evaluating all the attribute values for them. Unfortunately, the former subclasses are too small for describing the dependency between the attributes of the practical programming languages.

2.2.3 Transitive and reflexive closure

In this section, we shall present some algorithms for computing the transitive closure of a binary relation (or, for a directed graph).

Definition 2.2.13 *A* **(binary) relation** R *on the set* X *is a collection of ordered pairs of elements of* X*, i.e.* $R \subseteq X \times X$*. If* $(x_i,\ x_j) \in R$*, where* $x_i,\ x_j \in X$ *then we say that* x_i *and* x_j *are in the relation* R *(this can also be denoted by* $x_i\, R\, x_j$*).*

A convenient way of representing a binary relation R on a set X is to use a directed graph, the vertices of which stand for the elements of X and the arcs stand for the ordered pairs of elements of X defining the relation R.

Definition 2.2.14 *Consider a set* X *and a relation* R *on* X*. Then:*

1. *R is* **reflexive** *if every element $x_i \in X$ is in relation R to itself; that is, for every x_i, $x_i \, R \, x_i$;*

2. *R is* **symmetric** *if $x_i \, R \, x_j$ implies $x_j \, R \, x_i$;*

3. *R is* **transitive** *if $x_i \, R \, x_j$ and $x_j \, R \, x_k$ imply $x_i \, R \, x_k$;*

4. *R is an* **equivalence relation** *if it is reflexive, symmetric, and transitive. If R is an equivalence relation defined on a set S, then S can be uniquely partitioned into subsets S_1, S_2, ... such that two elements x and y of S belong to S_i if and only if $x \, R \, y$. The subsets S_1, S_2, ... are called the* **equivalence classes** *induced by the relation R on the set S.*

Definition 2.2.15 *The directed graph representing a reflexive relation is called the* **reflexive directed graph. Symmetric** *and* **transitive directed graph** *may be defined in a similar way.*

Remark 2.2.1

1. *In a reflexive directed graph, there exists a (self-)loop at each vertex.*

2. *In a symmetric directed graph, there exist two oppositely oriented arcs between any two adjacent vertices. Therefore an undirected graph can be considered as representing a symmetric relation if we identify any two oppositely oriented arcs with an arc.*

3. *The arc (v_1, v_2) also occurs in a transitive graph G if there exists a directed path in G from v_1 to v_2.*

Definition 2.2.16 *The* **transitive closure** *of a binary relation R is a relation R^+ defined as follows: $x \, R^+ \, y$ if and only if there exists a sequence*

$$x_0 = x, \; x_1, \; x_2, \; ..., \; x_k = y$$

such that $k > 0$ and $x_0 \, R \, x_1$, $x_1 \, R \, x_2$, ..., $x_{k-1} \, R \, x_k$.

Clearly, if $x \, R \, y$, then $x \, R^+ \, y$. Hence $R \subseteq R^+$. Further, it can be easily shown that R^+ is transitive. In fact, it is the smallest transitive relation containing R. So if R is transitive, then $R^+ = R$.

Definition 2.2.17 *Suppose that G is the directed graph representing a relation R. The directed graph G^+ representing the transitive closure R^+ of R is called the* **transitive closure** *of G.*

It follows from the definition of R^+ that the arc (x, y), $x \neq y$, is in G^+ if and only if there exists in G a directed path from the vertex x to the vertex y. Similarly the self-loop (x, x) at vertex x is in G^+ if and only if there exists in G a directed circuit containing x.

Definition 2.2.18 *The* **adjacency matrix** *of an $n-$vertex directed graph G is an $n \times n$ $(0,1)$ matrix in which the (i,j) entry is equal to 1 if and only if there exists a directed arc from vertex i to vertex j when $i \neq j$.*

The **reachability matrix** *of an $n-$vertex directed graph G is an $n \times n$ $(0,1)$ matrix in which the (i,j) entry is equal to 1 if and only if there exists a directed path from vertex i to vertex j when $i \neq j$, or a directed circuit containing vertex i when $i = j$.*

In other words, the (i,j) entry of the reachability matrix is equal to 1 if and only if vertex j is reachable from vertex i through a sequence of directed arcs. It is easy to see that the adjacency matrix of G^+ is the same as the reachability matrix of G.

The problem of constructing the transitive closure of a directed graph arises in several applications concerning compilers and attribute grammars. In this section, we point out only;

- an elegant and computationally efficient algorithm due to S. Warshall ([War62]) for computing the transitive closure;
- a variation of Warshall's algorithm given by H. S. Warren ([War75]).

Let G be an $n-$vertex directed graph with its vertices denoted by the integers 1, 2, ..., n. Let $G^0 = G$. Warshall's algorithm constructs a sequence of graphs so that $G^i \subseteq G^{i+1}$, $0 \leq i \leq n-1$, and G^n is the transitive closure of G. The graph G^i, $i \geq 1$, is obtained from G^{i-1} by "processing" vertex i in G^{i-1}. Processing the vertex i in G^{i-1} involves the addition of new edges to G^{i-1} as described below.

Let the arcs (i,k), (i,l), (i,m), ... of G^{i-1} be (outer) incident with the vertex i. Then for each arc (j,i) inner incident with the vertex i, add to G^{i-1} the arcs (j,k), (j,l), (j,m), ... if these arcs are not already present in G^{i-1}. The resulting graph (after the vertex i is processed) is denoted as G^i. It is clear that $G^i \subseteq G^{i+1}$, $\forall\, i \geq 0$. To show that G^n is the transitive closure of G we need to prove the following result.

Theorem 2.2.2 *1. Suppose that, for any two vertices s and t, there exists in G a directed path P from vertex s to vertex t such that all its vertices other than s and t are from the set $\{1,\ 2,\ ...,\ i\}$. Then G^i contains the arc (s,t).*

2. Suppose that, for any vertex s, there exists in G a directed circuit C containing vertex s such that all its vertices other than vertex s are from the set $\{1,2,\ ...,\ i\}$. Then G^i contains the self-loop (s,s).

Proof Following [ThS92]. ■

As an immediate consequence of this theorem we get the following.

Corollary 2.2.1 *G^n is the transitive closure of G.*

We give now a formal description of Warshall's algorithm. In this description the graph G is represented by its adjacency matrix M and the symbol $\vee$ stands for Boolean addition ($a \vee b = 1 \iff a = 1$ *or* $b = 1$).

Algorithm (TC1). Transitive Closure (S. Warshall)

Input: M, the adjacency matrix of a graph G;
Output: M^+, the reachability matrix of G;
Method:

```
begin
  for i := 1 to n do
    for j := 1 to n do
      if M(j,i) = 1 then
        for k := 1 to n do
          M(j,k) := M(j,k) ∨ M(i,k)
  M⁺ = M
end.
```

Note that the matrix M (when the algorithm begins to execute the statement **for** $j := 1$ **to** n **do** with $i = p$) is the adjacency matrix of G^{i-1}. Further, on processing a diagonal entry does not result in adding new nonzero entries.

Let us note that:

1. Warshall's algorithm transforms the adjacency matrix M of a graph G into the adjacency matrix of the transitive closure of G by suitable overwriting on M. For this reason we say that the algorithm works "in place".

2. The algorithm processes all the arcs incident into a vertex before it begins to process the next vertex. In other words it processes the matrix M "column-wise". Warshall's algorithm may be thus seen as *column-oriented*.

3. While processing a vertex, no new arc (i.e., an arc that does not exists when the processing of that vertex begins) inner incident to the vertex is added to the graph. This means during the processing of a vertex we can choose the arcs inner incident to that vertex in an arbitrary order.

4. Suppose that the arc (j,i) inner incident into the vertex i during the processing of the vertex i and it is added in a further step, during the processing some vertex k, $k > i$. Clearly this arc is not processed during the processing of the vertex i. Neither will it be processed later since no vertex is processed more than once. In fact, such an arc will not result in adding any new arcs.

5. Warshall's algorithm is said to work in one pass since each vertex is processed exactly once.

Suppose that we wish to modify Warshall's algorithm so that it becomes *row-oriented*. In a row-oriented algorithm, during processing a vertex, all the arcs outer incident to that vertex are to be processed. The processing of the arc (i, j) introduces the arcs (i, k) for every arc (j, k) outer incident to the vertex j. Therefore new arcs outer incident to a vertex may be added during the processing of a vertex "row-wise". Some of these newly added arcs may not be processed before the processing of the vertex under consideration is completed. If the processing of these arcs is necessary for the computation of the transitive closure, then such a processing can be done only in a second pass. Thus, a row-oriented algorithm may require more than one pass to compute the transitive closure.

H. S. Warren ([War75]) proved that only two passes in the row-oriented algorithm are really needed. During the processing of vertex i, in the first pass only arcs connected to vertices less than i are processed, and in the second pass only arcs connected to vertices greater than i are processed. In other words, the algorithm transforms the adjacency matrix of G^+ by processing in the first pass only entries below the main diagonal of M and in the second pass only entries above the main diagonal. Thus during each pass at most $n(n-1)/2$ arcs are processed.

Algorithm (TC2). Transitive Closure (Warren)

Input: M, the adjacency matrix of a graph G;
Output: M^+, the reachability matrix of G;
Method:

```
begin { Part 1 }
    for i := 2 to n do
      for j := 1 to i - 1 do
        if M(i, j) = 1 then
          for k := 1 to n do
            M(i, k) := M(i, k) ∨ M(j, k)
    { Part 2 }
    for i := 1 to n - 1 do
      for j := i + 1 to n do
        if M(i, j) = 1 then
          for k := 1 to n do
            M(i, k) := M(i, k) ∨ M(j, k)
    M+ = M
end.
```

The proof of the correctness of Warren's algorithm is based on the following lemma.

Lemma 2.2.3 *Suppose that, for any two vertices s and t, there exists in G a directed path P from s to t. Then the graph that results after processing vertex s in the first pass (the statements included in Part 1) of Warren's algorithm contains an arc (s, r), where r is a successor of s on P and either $r > s$ or $r = t$.*

Proof Following [ThS92]. ■

Theorem 2.2.3 *Warren's algorithm computes the transitive closure of a graph G.*

Proof Following [ThS92]. ■

Both Warshall's and Warren's algorithms have the (worst-case) complexity $\mathcal{O}(n^3)$, where n is the cardinality of X. Other row-oriented algorithms may be found in [War75].

2.3 Parallel computers

A *parallel computer* is a computer with many processing units (processors). A *parallel algorithm* is a solution method for a given problem destinate to be solved (performed) on a parallel computer.

2.3.1 Models of (parallel) computation

Any computer, whether sequential or parallel, operates by executing instructions on data. A stream of instructions (the algorithm) tells the computer what to do at each step. A stream of data (the input of the algorithm) is affected by these instructions. Depending on whether there is one or several of these streams, we can distinguish among four classes of computers (type of computations):

1. Single Instruction stream, Single Data stream (SISD)
2. Multiple Instruction stream, Single Data stream (MISD)
3. Single Instruction stream, Multiple Data stream (SIMD)
4. Multiple Instruction stream, Multiple Data stream (MIMD)

2.3.1.1 SISD computers

A computer in this class consists of a single processing unit receiving a single stream of instructions that operates on a single stream of data. At each step during the computation the control unit emits one instruction that operates on a datum obtained from the memory unit. Such an instruction may tell the processor, for example, to perform some arithmetic or logic operation on the datum and then put it back into the memory.

The overwhelming majority of computers today adhere to this model invented by John von Neumann and his collaborators in the late 1940's. An algorithm for a computer in this class is said to be *sequential* (or *serial*).

Example 2.3.1 *In order to compute the sum of n numbers, the unique processor needs to gain access to the memory n consecutive units of time at each time receiving one number. There are also $n-1$ additions involved that are executed in sequence. Therefore, this computation requires $\mathcal{O}(n)$ operations.*

2.3.1.2 MISD computers

In this case, N processors (each with its own control unit) share a common memory unit where data reside. There exist N streams of instructions and one stream of data. At each step, one datum received from memory is operated upon by all the processors simultaneously, each according to the instruction it receives from its control. Thus, parallelism is achieved by letting the processors do different things at the same time on the same datum. This class of computers lends itself naturally to those computations requiring an input to be subjected to several operations, each receiving the input in its original form.

Example 2.3.2 *It is required to determine whether a given positive integer z has no divisors except* 1 *and itself. The obvious solution to this problem is to try all possible divisors of z. If none of these succeeds in dividing z, then z is said to be* prime*; otherwise z is said to be* composite*.*

We can implement this solution as a parallel algorithm on an MISD computer. The idea is to split the job of testing potential divisors among processors. Assume that there are as many processors on the parallel computer as there are potential divisors of z. All processors take z as input, then each tries to divide it by its associated potential divisor and issues an appropriate output based on the result. Thus it is possible to determine in one step *whether z is prime. More realistically, if there are fewer processors than potential divisors, then each processor can be given the job of testing a different subset of these divisors. In either case, a substantial speedup is obtained over a purely sequential implementation.*

Although more efficient solutions to the problem of primality testing *exist, we have chosen the simplest one.*

Example 2.3.2 shows that the class of MISD computers could be extremely useful in many applications. It is also apparent that the kind of computations that can be carried out efficiently on these computers are of a rather specialized nature. For most applications, MISD computers would be rather awkward to use. We shall see that SIMD and MIMD are more suitable for a wide range of problems.

2.3.1.3 SIMD computers

A parallel computer in this class consists of N identical processors, as shown in Figure 2.4.

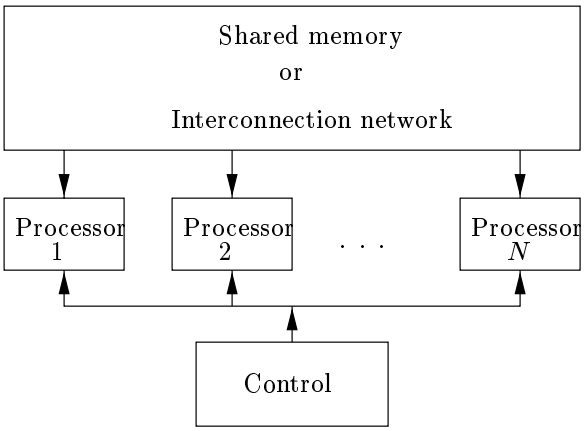


Figure 2.4. SIMD Computer

Each of the N processors possesses its own local memory where it can store both programs and data. All processors operate under the control of a single instruction stream issued by a central control unit. Equivalently, the N processors may be assumed to hold identical copies of a single program, each processor's copy being stored in its local memory. There are N data streams, one per each processor.

The processors operate synchronously. At each step, all processors execute the same instruction, each on a different datum. The *instruction* could be a simple one (such as adding or comparing two numbers) or a complex one (such as merging two lists of numbers). Similarly, the *datum* may be simple (one number) or complex (several numbers). Sometimes, it may be necessary to have only a subset of the processors executing an instruction. This information can be encoded in the instruction itself, thereby telling a processor whether it should be *active* (and execute the instruction) or *inactive* (and wait for the next instruction). There is a mechanism, such as *a global clock*, that ensures lock-step operation. Thus processors that are inactive during an instruction or those that complete the execution of the instruction before others may stay idle until the next instruction is issued. The time interval between two instructions may be fixed or may depend on the instruction being executed.

In most interesting problems that we wish to solve on an SIMD computer, it is desirable for the processors to be able to communicate among themselves during the computation in order to exchange data or intermediate results. This can be achieved in two ways, giving rise to two subclasses: SIMD computers where communication is through a *shared memory* and those where communication is done via an *interconnection network*.

Shared-Memory (SM) SIMD computers

This class is also known in the literature as the Parallel Random-Access Machine (PRAM) model. When two processors wish to communicate, they do so through the shared memory. Say processor i wishes to pass a number to processor j.

This is done in two steps. First, processor i writes the number in the shared memory at a given location known to processor j. Then, processor j reads the number from that location.

During the execution of a parallel algorithm, the N processors gain access to the shared memory for reading input data, for reading or writing intermediate results, and for writing final results. The basic model allows all processors to gain access to the shared memory simultaneously if and only if the memory locations they are trying to read from or write into are different. However, the class of shared-memory SIMD computers can be further divided into four subclasses, according to whether two or more processors can gain access to the same memory location simultaneously:

(i) **Exclusive-Read, Exclusive-Write (EREW) SM SIMD Computers**. Access to memory locations is exclusive. In other words, no two processors are allowed simultaneously to read from or write into the same memory location.

(ii) **Concurrent-Read, Exclusive-Write (CREW) SM SIMD Computers**. Multiple processors are allowed to read from the same memory location but the right to write is still exclusive.

(iii) **Exclusive-Read, Concurrent-Write (ERCW) SM SIMD Computers**. Multiple processors are allowed to write into the same memory location but read accesses remain exclusive.

(iv) **Concurrent-Read, Concurrent-Write (CRCW) SM SIMD Computers**. Both multiple-read and multiple-write privileges are granted.

Allowing multiple-read accesses to the same address in memory generates no problems (except perhaps some technological ones to be discussed later). Conceptually, each of the several processors reading from that location makes a copy of the location's contents and stores it in its own local memory.

With multiple-write accesses, however, difficulties arise. If several processors are attempting simultaneously to store (potentially different) data at a given address, which one of them should succeed ? In other words, there should be a deterministic way of specifying the contents of that address after the write operation. Several policies have been proposed to resolve such *write conflicts*, thus further subdividing classes (iii) and (iv). Some of these policies are:

(a) the smallest-numbered processor is allowed to write, and access is denied to all other processors;

(b) all processors are allowed to write provided that the quantities they are attempting to store are equal, otherwise access is denied to all processors;

(c) the sum of all quantities that the processors are attempting to write is stored.

Interconnection-Network SIMD computers

There exists a lot of interconnection possibilities between processors for a SIMD computer. One of the simplest interconnection is that where every pair of processors are connected by a two-way line, i.e. a structure of a complete graph. Unfortunately, this model is not at all convenient in practice, because:

(i) **Price.** What is the price paid to fully interconnect N processors ? There are $N-1$ lines leaving each processor for a total of $\frac{N(N-1)}{2}$. Clearly, such a network is too expensive, especially for large values of N.

(ii) **Feasibility.** Even if we could afford such a high price, the model is unrealistic in practice, again for large values of N. Indeed, there is a limit on the number of lines that can be connected to one processor, and that limit is dictated by the actual physical size of the processor itself.

Other possibilities for interconnecting processors are: *linear array, two-dimensional array, tree connection, perfect shuffle connection, cube connection.*

Example 2.3.3 *Assume that the sum of n numbers x_1, x_2, ..., x_n needs to be computed. There exist $n-1$ additions involved in this computation, and a sequential algorithm running on a conventional (i.e. SISD) computer will require n steps to complete it. Using a tree-connected SIMD computer with $\log n$ levels and $\frac{n}{2}$ leaves, the job can be accomplished in $\log n$ steps (Figure 2.5.) for $n = 8$.*

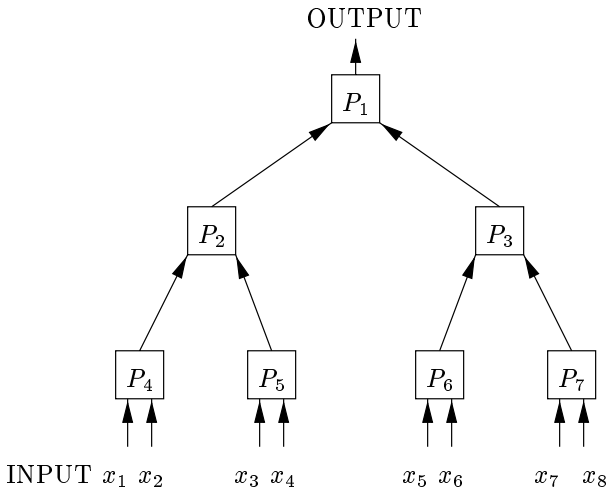


Figure 2.5. Adding eight numbers on a processor tree

The original input is received at the leaves, two numbers per leaf. Each leaf adds its inputs and sends the result to its parent. The process is now repeated at each subsequent level: Each processor receives two inputs from its children, computes their sum, and sends it to its parents. The final result is eventually

produced by the root. Since at each level all the processors operate in parallel, the sum is computed in $\log n$ *steps.*

The speed improvement is even more dramatic when m *sets, each of* n *numbers, are available and the sum of each set is to be computed. A conventional (sequential) machine requires* $m \cdot n$ *steps in this case. A trivial application of the parallel algorithm produces the* m *sums in* $m(\log n)$ *steps. Through a process known as* pipelining*, however, we can do it significantly better. Notice that once a set has been processed by the leaves, these are free to receive the next one. The same observation applies to all processors at the higher levels. Hence each of the* $m-1$ *sets that follow the initial one can be inputed to the leaves one step after their predecessor. Once the first sum exits from the root, a new sum is produced in the next step. The entire process therefore takes* $\log n + (m-1)$ *steps.*

2.3.1.4 MIMD computers

This class of computers is the most general and the most powerful. A MIMD computer has N processors, N streams of instructions, and N streams of data. The processors here are of the type used in MISD computer in the sense that each possesses its own control unit in addition to its local memory and arithmetical and logical units. This makes these processors more powerful than the ones used for SIMD computers.

Each processor operates under the control of an instruction stream issued by its control unit. Thus the processors are (potentially all) executing different programs on different data while solving different subproblems of a single problem. This means that the processors typically operate asynchronously. As with SIMD computers, communication between processors is performed through a shared memory or an interconnection network. MIMD computers sharing a common memory are often referred to as *multiprocessors* (or *tightly coupled machines*) while those with an interconnection network are known as *multicomputers* (or *loosely coupled machines*).

Since all the processors on a multiprocessor computer share a common memory, the discussion in section 2.3.1.3 regarding the various ways of concurrent memory access applies here as well. Indeed, two or more processors executing an (asynchronous) algorithm may, by accident or by design, wish to gain access to the same memory location. We can therefore talk of EREW, CREW, ERCW, and CRCW SM MIMD computers and algorithms, and various methods should be established for solving memory access conflicts in models that disallow them.

Computers in the MIMD class are used to solve those problems that lack the regular structure required by the SIMD model. This generality does not come for free, that is asynchronous algorithms are difficult to design, evaluate, and implement. In order to appreciate the complexity involved in programming MIMD computers, it is important to distinguish between the notion of a *process* and that of a *processor*. An asynchronous algorithm is a collection of processes some or all of which are executed simultaneously on a number of (available) processors. At the beginning (at the beginning of the execution), all the processors are free. The parallel algorithm starts its execution on an

arbitrarily chosen processor(s). Shortly thereafter it creates a number of computational tasks (processes) to be performed. A process thus corresponds to a section of the algorithm: There may be several processes associated with the same algorithm section, each with a different parameter.

Once a process is created, it must be executed on a processor. If a free processor is available, the process is assigned to the processor that performs the computations specified by the process. Otherwise (if no free processor is available), the process is queued and waits for a processor to be free.

When a processor completes execution of a process, it becomes free. If a process is waiting to be executed, then it can be assigned to the processor just freed. Otherwise (if no process is waiting), the processor is queued and waits for a process to be created.

The order in which processes are executed by processors can obey any policy that assigns, e.g., priorities to processes. For example, processes can be executed in a first-in-first-out or in a last-in-first-out order. Also, the availability of a processor is sometimes not sufficient for the processor to be assigned to a waiting process. An additional condition may have to be satisfied before the process starts. Similarly, if a processor has already been assigned a process and an unsatisfied condition is encountered during execution, then the processor is freed. When the condition for resumption of that process is later satisfied, a processor (not necessarily the original one) is assigned to it. These are only a few of the scheduling problems that may characterize the programming of multiprocessors. Finding efficient solutions to these problems is important if MIMD computers are to be considered useful. Note that none of these scheduling problems arise on the less flexible but easier to program SIMD computers.

2.3.2 Analyzing algorithms

Related to the analysis of parallel algorithms, the most important measure is the *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. 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.

Example 2.3.4 *The* sorting problem *is defined as follows: A set of n (real) numbers is given has to be sorted in the nondecreasing order. Note that there exist $n!$ possible permutations of the input and $\log n!$ (i.e. $\mathcal{O}(n \log n)$) bits are*

needed to distinguish among them. Therefore, in the worst case, any sequential algorithm for sorting requires on $\mathcal{O}(n \log n)$ steps to recognize a particular output (as described below).

Example 2.3.5 *Say that we want to compute the product of two $n \times n$ matrices. Since the resulting matrix has also n^2 entries, at least n^2 steps are needed by any matrix multiplication algorithm simply to produce the output. Up to now, no algorithm is known for multiplying two $n \times n$ matrices in n^2 steps. The standard textbook algorithm requires $\mathcal{O}(n^3)$ operations.*

For parallel algorithms, two additional factors (apart of the lower and upper bounds) have to be taken into consideration:

(i) the model of parallel computation used;

(ii) the number of processors involved.

To evaluate 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. A good indication of the quality of a parallel algorithm is the *speedup* it produces. This is defined as

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

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

Example 2.3.6 *The problem of adding n numbers discussed in Example 2.3.3 is solved in $\mathcal{O}(\log n)$ time on a tree-connected parallel computer using $n-1$ processors. Here the speedup is $\mathcal{O}(n/\log n)$ since the best possible sequential algorithm requires $\mathcal{O}(n)$ additions. This speedup is far from the ideal $n-1$ and it is due to the fact that the n numbers were input at the leaves and the sum outputs at the root. Any algorithm for such a model necessarily requires $\Omega(\log n)$ time, that is, the time required for a single datum to propagate from input to output through all levels of the tree.*

The second important criterion in evaluating a parallel algorithm is the **number of processors** it requires to solve a problem. The larger the number of processors an algorithm uses to solve a problem, the more expensive the solution becomes to be obtained. 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 may be 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, the 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 the 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 possible steps. It may be however 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.

Example 2.3.7 *In Example 2.3.3, the size of the tree depends on n, the number of terms to be added, and $p(n) = n - 1$. The cost of adding n numbers on an $(n-1)$-processor tree is $(n-1) \times \mathcal{O}(\log n)$. This cost is not optimal since we know how to add n numbers optimally using $\mathcal{O}(n)$ sequential additions.*

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.

Remark 2.3.1 *Since $\Omega(n \log n)$ steps is a lower bound on any sequential sorting algorithm, the equivalent lower bound on any parallel algorithm using n processors is $\Omega(\log n)$.*

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:

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

Usually, the *efficiency* has to be less than or equal to 1; otherwise a faster sequential algorithm can be obtained !

Example 2.3.8 *Let the worst-case running time of the fastest sequential algorithm for multiplying two $n \times n$ matrices be $\mathcal{O}(n^{2.5})$ (as known). The efficiency of a parallel algorithm that uses n^2 processors to solve the problem in $\mathcal{O}(n)$ time is $\mathcal{O}(n^{2.5})/\mathcal{O}(n^3)$.*

Chapter 3

Linear-time bidirectional parsing for a subclass of linear grammars

This chapter is the first original one from this paper. It was prepared using the papers [AnK98] and [AnK99b].

New classes of linear grammars, $LLin(m,n)$, $m,n \in \mathbf{N}$ - similar to the $LL(k)$, $k \in \mathbf{N}$ ([AhU72], [LeS68]) grammars - were introduced ([AnK98, AnK99b]). Intuitively, "looking ahead" to the next m terminal symbols and "looking back" to the previous n terminal symbols suffices to uniquely determine the production which has to be applied. The membership problem for $LLin(m,n)$ grammars can be solved using a linear time complexity algorithm.

In the first section we give some general properties of the mentioned grammars, such as unambiguity, recursiveness and closure properties. A comparison with $LL(k)$ grammars and an "internal" hierarchy is also provided. Let us note that there exist $m,n \in \mathbf{N}$ such that $\mathcal{LL}in(m,n)$ are strictly between deterministic context free languages and context free languages.

In the second section, a characterization theorem for $LLin(m,n)$ grammars is presented. We also describe a bidirectional parser for $LLin(m,n)$ grammars.

The third section treats $LLin(1,1)$ grammars. One of the main point is that the auxiliary function $first\_last$ can be computed in polynomial time. In this way, we can easily decide whether or not a linear grammar is $LLin(1,1)$.

3.1 Definitions and general properties

In this chapter we introduce a new subclass of linear languages for which the membership problem can be solved in linear time (sequential) complexity. For the general class of linear languages, it is known that an arbitrary word w with length n can be parsed in time proportional to n^2 ([Har78]). We know that

every sentential form of a linear grammar contains at most one nonterminal symbol. Using this property, our subclass of linear grammars can be view as a generalization of $LL(k)$ grammars. The difference is that for the new subclass the parsing is simultaneously done from both sides of the word.

Definition 3.1.1 *Let $G = (V_N, V_T, S, P)$ be a linear grammar. We say that G is $LLin(m,n)$, $m,\ n \in \mathbf{N}$, if for any two derivations of the form*

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_1\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,x\,v$$

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_2\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,y\,v$$

with $u,\ v,\ x,\ y \in V_T^*$, for which $x^{(n)} = y^{(n)}$ and ${}^{(m)}x = {}^{(m)}y$, then $\beta_1 = \beta_2$.

Intuitively: *Given an arbitrary sentential form, if we "look back" to the previous n terminal symbols and if we "look ahead" to the next m terminal symbols, we can uniquely decide which production has to be applied* (Figure 3.1) (the overlapping of symbols is allowed).

Definition 3.1.2 *We say that the language $L \subseteq V_T^*$ is $\mathcal{LL}in(m,n)$ if there exists a $LLin(m,n)$ grammar G for which $L = L(G)$.*

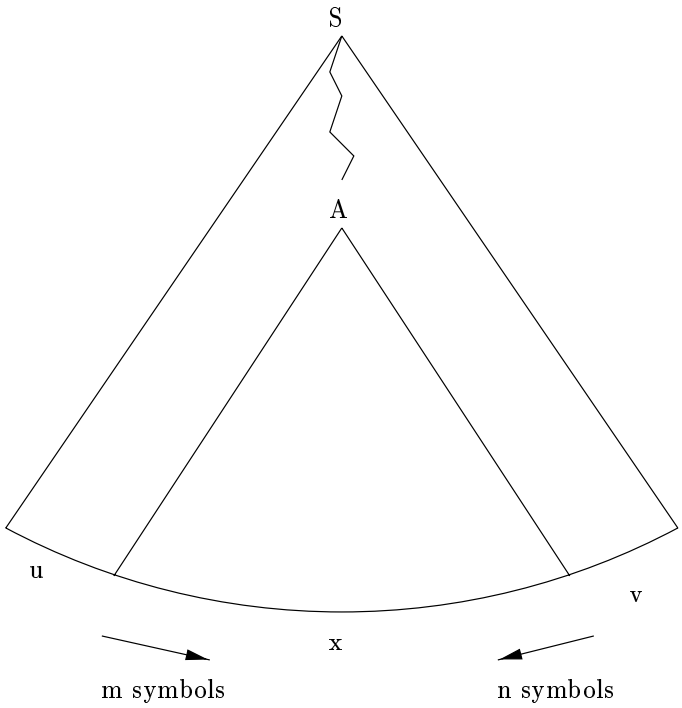


Figure 3.1

In the next example we give some representative linear languages which can be expressed using $LLin(m,n)$ grammars.

Example 3.1.1

- $G_1 = (\{S\}, \{a,b,c\}, S, \{S \to a\,S\,a \mid b\,S\,b \mid c\})$ *is* $LLin(1,1)$ *and, of course,* $L(G_1) = \{w\,c\,\widetilde{w} \mid w \in \{a,b\}^*\}$;
- $G_2 = (\{S\}, \{a,b,c\}, S, \{S \to a\,S\,a \mid a\,S\,b \mid b\,S\,a \mid b\,S\,b \mid c\})$ *is* $LLin(1,1)$ *and* $L(G_2) = \{w_1\,c\,w_2 \mid w \in \{a,b\}^*,\ |w_1| = |w_2|\}$;
- $G_3 = (\{S\}, \{a,b,c\}, S, \{S \to a\,a\,S\,a\,a \mid a\,b\,S\,a\,b \mid a\,b\,S\,b\,a \mid b\,a\,S\,a\,b \mid b\,a\,S\,b\,a \mid b\,b\,S\,b\,b \mid c\})$ *is* $LLin(2,2)$ *and* $L(G_3) = \{w_1\,c\,w_2 \mid w \in \{a,b\}^*,\ |w_1| = |w_2| =$ *even,* $N_{w_1}(a) = N_{w_2}(a)$ *and* $N_{w_1}(b) = N_{w_2}(b)\}$, *where* $N_{w_1}(a)$ *denotes the number of symbols 'a' from* w_1;
- $G_4 = (\{S,A\}, \{a,b\}, S, \{S \to a\,S \mid A\,b,\ A \to A\,b \mid \lambda\})$ *is* $LLin(1,1)$ *and* $L(G_4) = \{a^n\,b^m \mid n \geq 0,\ m \geq 1\}$;
- $G_5 = (\{S,A\}, \{a,b,c\}, S, \{S \to a\,S\,a \mid A,\ A \to b\,A\,b \mid c\})$ *is* $LLin(1,1)$ *and* $L(G_5) = \{a^n\,b^m\,c\,b^m\,a^n \mid n,\ m \geq 1\}$;
- $G_6 = (\{S,A,B\}, \{a,b\}, S, \{S \to A\,c \mid B,\ A \to a\,A\,b\,b \mid a\,b\,b,\ B \to a\,B\,b \mid a\,b\})$ *is* $LLin(2,1)$ *and* $L(G_6) = \{a^n\,b^{2n}\,c,\ a^n\,b^n \mid n \geq 1\}$;

A linear grammar may be ambiguous. For example, let us consider the linear grammar G given by the productions:

1. $S \to a\,S$
2. $S \to S\,a$
3. $S \to a$

For the word $w = a\,a\,a$, there exist two left most (or right most) derivations:

$$S \underset{G}{\Longrightarrow} a\,S \underset{G}{\Longrightarrow} a\,S\,a \underset{G}{\Longrightarrow} a\,a\,a$$

$$S \underset{G}{\Longrightarrow} S\,a \underset{G}{\Longrightarrow} a\,S\,a \underset{G}{\Longrightarrow} a\,a\,a$$

Hence G is an ambiguous linear grammar.

We shall show that the subclass $LLin(m,n)$ contains only unambiguous grammars.

Theorem 3.1.1 *Every* $LLin(m,n)$ *grammar is unambiguous.*

Proof Let $G = (V_N, V_T, S, P)$ be an $LLin(m,n)$ grammar and suppose that it is ambiguous. Then there exists a word $w \in L(G)$ such that we can construct two distinct derivations (in G):

$$S = \alpha_0 \underset{lm}{\Longrightarrow} \alpha_1 \underset{lm}{\Longrightarrow} \alpha_2 \underset{G}{\Longrightarrow} \ldots \underset{lm}{\Longrightarrow} \alpha_k = w$$

$$S = \beta_0 \underset{lm}{\Longrightarrow} \beta_1 \underset{lm}{\Longrightarrow} \beta_2 \underset{G}{\Longrightarrow} ... \underset{lm}{\Longrightarrow} \beta_{k'} = w$$

We shall show by induction on i that $\alpha_i = \beta_i,\ \forall\ i \geq 0$. The basis of induction ($i = 0$) is immediate.

Let us suppose that $\alpha_j = \beta_j,\ \forall\ 0 \leq j \leq i$. We have to prove that $\alpha_{i+1} = \beta_{i+1}$. Because G is a linear grammar (all the derivations are left most and also right most), we can rewrite the previous derivation as:

$$S \underset{G}{\overset{*}{\Longrightarrow}} \alpha_i = u\,A\,v \underset{G}{\Longrightarrow} u\,\gamma_1\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,x\,v = w$$

$$S \underset{G}{\overset{*}{\Longrightarrow}} \beta_i = u\,A\,v \underset{G}{\Longrightarrow} u\,\gamma_2\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,y\,v = w$$

From $u\,x\,v = w$ and $u\,y\,v = w$, it follows that $x = y$. Therefore, ${}^{(m)}x =^{(m)} y$ and $x^{(n)} = y^{(n)}$. Thus $\gamma_1 = \gamma_2$; so $\alpha_{i+1} = u\,\gamma_1\,v = u\,\gamma_2\,v = \beta_{i+1}$. Hence, $\alpha_i = \beta_i,\ \forall\ 0 \leq i \leq \min(k, k')$. But $\alpha_k = \beta_{k'} = w$, so $k = k'$. Therefore the assumption that G is ambiguous is false. ■

We shall denote the set of $LL(k)$ linear grammars by $LLin(k)$. We may easily observe that the class of $LLin(1,1)$ grammars is larger than the class of $LLin(1)$ grammars. For instance, G_2 (Example 3.1.1) is $LLin(1,1)$, but not $LLin(1)$.

Lemma 3.1.1 *Every $LLin(k)$ grammar is a $LLin(k, k')$ grammar, $\forall\ k' \geq 0$.*

Proof Directly from the definitions. ■

Lemma 3.1.2 *If G is a $LLin(m, n)$ grammar, then G is a $LLin(m', n')$ grammar, where $m' \geq m,\ n' \geq n$.*

Proof Directly from the definitions. ■

Theorem 3.1.2 *For all $m, n \geq 0$, the class $LLin(m, n)$ is properly included in the class $LLin(m', n'),\ \forall\ m' \geq m,\ \forall n' \geq n$.*

Proof The fact that $LLin(m, n)$ is included in $LLin(m', n')$ is obvious, where $m' \geq m,\ n' \geq n$ (Lemma 3.1.2). It remains to show that the inclusion is proper.

Let us consider the following linear grammar:

$$G:\ S \rightarrow a^m\,b^n \mid a^{m'}\,b^{n'}\ \ (m' \geq m,\ n' \geq n)$$

It is obvious that G is $LLin(m', n')$, but not $LLin(m, n)$ (of course, we have $(m' - m)^2 + (n' - n)^2 \neq 0$). ■

Theorem 3.1.3 *There exist linear languages which are not $\mathcal{LL}in(m, n)$, for any $m,\ n \in \mathbf{N}$.*

Proof Let us consider the linear language $L = L_1 \cup L_2$, where

$$L_1 = \{a^k\,c\,b^k \mid k \geq 1\} \text{ and } L_2 = \{a^k\,d\,b^{2k} \mid k \geq 1\}.$$

For instance, L can be generated by the linear grammar G_3:

- $S \to A \mid B$
- $A \to a\,A\,b \mid c$
- $B \to a\,B\,b\,b \mid d$

Let us suppose, on the contrary, that there exist $m, n \in \mathbf{N}$ and $G \in LLin(m,n)$ such as $L(G) = L$. Let us denote $i = \max(m,n)$ and the words $w_1 = a^i\,c\,b^i$, $w_2 = a^i\,d\,b^{2i}$ which belong to L_1, respectively L_2. Because $L = L(G)$, then there exist the derivations:

$$S \underset{G}{\overset{*}{\Longrightarrow}} a^i\,c\,b^i$$

$$S \underset{G}{\overset{*}{\Longrightarrow}} a^i\,d\,b^{2i}$$

It is obvious that ${}^{(m)}w_1 =^{(m)} w_2$ and $w_1^{(n)} = w_2^{(n)}$. This means that the last production applied in the above derivations is the same. Let $a^k\,A\,b^j$ be the last sentential form for which:

$$S \underset{G}{\overset{*}{\Longrightarrow}} a^k\,A\,b^j \underset{G}{\Longrightarrow} a^k\,\beta_1\,b^j \underset{G}{\overset{*}{\Longrightarrow}} a^k\,a^{i-k}\,c\,b^{i-j}\,b^j = w_1$$

$$S \underset{G}{\overset{*}{\Longrightarrow}} a^k\,A\,b^j \underset{G}{\Longrightarrow} a^k\,\beta_2\,b^j \underset{G}{\overset{*}{\Longrightarrow}} a^k\,a^{i-k}\,d\,b^{2i-j}\,b^j = w_2$$

and ${}^{(m)}a^{i-k}\,c\,b^{i-j} =^{(m)} a^{i-k}\,d\,b^{2i-j}$, $a^{i-k}\,c\,b^{i-j}{}^{(n)} = a^{i-k}\,d\,b^{2i-j}{}^{(n)}$. Because G is $LLin(m,n)$ it follows that $\beta_1 = \beta_2$. During the derivation $a^k\,A\,b^j \underset{G}{\overset{*}{\Longrightarrow}} a^k\,a^{i-k}\,c\,b^{i-j}\,b^j$ only productions corresponding to w_1 will be applied (which are distinct from productions corresponding to w_2, $w_1 \neq w_2$). So, we obtain a contradiction because $A \to \beta_1 = A \to \beta_2$. Therefore G is not a $LLin(m,n)$ grammar. ■

Corollary 3.1.1 *The following facts hold:*

- *G is $LLin(m,n)$ iff $\widetilde{G}$ is $LLin(n,m)$ (the class of $LLin(m,n)$ grammars is closed under mirror image, $\forall\ m \geq 0,\ n \geq 0$);*
- *G is $LLin(m,0)$ iff G is $LLin(m)$;*
- *G is $LLin(0,n)$ iff $\widetilde{G}$ is $LLin(n)$.*

Proof Directly from the definitions and the fact that $\widetilde{G}$ is also linear if G is a linear grammar. ■

It is known that a left-recursive grammar cannot be $LL(k)$ ([Knu65], [Knu71]), for any $k \geq 0$. However, there exist some procedures to transform left-recursion into right-recursion. On the contrary, there exist $LLin(m,n)$ left-recursive (even right-recursive, see G_4, Example 3.1.1) grammars. If a $LLin(m,n)$ grammar is both left and right recursive, this cannot come from the existence of the same left (right) recursive symbol.

Theorem 3.1.4 *If the reduced linear grammar G contains a (simultaneously) left and right recursive nonterminal symbol A, then G cannot be $LLin(m,n),\ \forall\ m,n \in \mathbf{N}$.*

Proof Let A be a left and right recursive (in the same time) symbol in G. Because G is linear, this means that there exist the derivations:

$$A \underset{G}{\overset{+}{\Longrightarrow}} A\,v',\ A \underset{G}{\overset{+}{\Longrightarrow}} u'\,A,\ u',\ v' \in V_T^+.$$

Without loss of generality, we may suppose that the first distinct productions applied in the above derivations are:

$$A \to B\,v_1 \text{ and respectively } A \to u_1\,C$$

Now, because G is also a reduced linear grammar, it follows that there exists a derivation:

$$S \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v$$

Now suppose that there exist $m,n \in \mathbf{N}$ such as G is $LLin(m,n)$. Continuing the above derivation, we may write:

$$S \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,x\,v_1\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v'\,v \underset{G}{\overset{+}{\Longrightarrow}} u\,u'\,A\,v'\,v \underset{G}{\overset{+}{\Longrightarrow}} \ldots \underset{G}{\overset{+}{\Longrightarrow}} u\,(u')^m A (v')^n\,v$$

$$S \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,u_1\,y\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,u'\,A\,v \underset{G}{\overset{+}{\Longrightarrow}} u\,u'\,A\,v'\,v \underset{G}{\overset{+}{\Longrightarrow}} .. \underset{G}{\overset{+}{\Longrightarrow}} u(u')^{m+1}A(v')^{n+1}v$$

But $^{(m)}\left((u')^m\,A\,(v')^n\right) =^{(m)} \left((u')^{m+1}\,A\,(v')^{n+1}\right)$ and $\left((u')^m\,A\,(v')^n\right)^{(n)} =$
$= \left((u')^{m+1}\,A\,(v')^{n+1}\right)^{(n)}$. Using the fact that G is $LLin(m,n)$, it follows that $A \to B\,v_1$ coincides with $A \to u_1\,C$ (a contradiction !).

Therefore G cannot be $LLin(m,n),\ \forall\ m,n \in \mathbf{N}$. ■

The elements of $\mathcal{LL}in(m,n)$ can generate some classical non-deterministic languages ([Knu65]), such as $L = \{a^n\,b^{2n}\,c,\ a^n\,b^n \mid n \geq 1\}$. For instance, G_6 (Example 3.1.1) can generate this language.

Theorem 3.1.5 *(closure properties) $\mathcal{LL}in(m,n)$ are not closed under:*

(i) union

(ii) intersection

(iii) catenation

(iv) homomorphism

Proof

(i) Let $G_1 = (\{S\},\{a,b,c\},S,\{S \to a\,S\,b \mid c\})$ and $G_2 = (\{S\},\{a,b,d\},S,\{S \to a\,S\,b\,b \mid d\})$ be two $LLin(1,0)$ grammars. We have $L(G_1) = \{a^k\,c\,b^k \mid k \geq 1\}$ and $L(G_2) = \{a^k\,d\,b^{2k} \mid k \geq 1\}$. The language $L(G_1) \cup L(G_2)$ is not a linear language (proof of Theorem 3.1.3);

(ii) Consider $G_1 = (\{S, A\}, \{a, b, c\}, S, \{S \to S\,c \mid A,\ A \to a\,A\,b \mid a\,b\})$ and $G_2 = (\{S, A\}, \{a, b, c\}, S, \{S \to a\,S \mid A,\ A \to b\,A\,c \mid b\,c\})$ be two $LLin(0,2)$ and $LLin(2,0)$ grammars, respectively. So $L(G_1) = \{a^n\,b^n\,c^m \mid m, n \geq 1\}$ and $L(G_2) = \{a^m\,b^n\,c^n \mid m, n \geq 1\}$. Then the intersection of these languages $L(G_1) \cap L(G_2) = \{a^n\,b^n\,c^n \mid n \geq 1\}$ is not a context free (or linear) language ([Har78], [HoU79], [JuA97]);

(iii) Let $G = (\{S\}, \{a, b\}, S, \{S \to a\,S\,b \mid a\,b\})$ be a $LLin(2,0)$ grammar. Certainly $L(G) = \{a^n\,b^n \mid n \geq 1\}$. We shall prove that the language $L = L(G) \cdot L(G) = \{a^n\,b^n\,a^m\,b^m \mid m, n \geq 1\}$ is not linear. Suppose, by contrary, that L is a linear language. From the pumping lemma for linear languages (Theorem 2.1.2), we can choose the word $z = a^N\,b^N\,a^N\,b^N \in L, N \in \mathbf{N}$. Then $u\,v \in \{a\}^*$ and $x\,y \in \{b\}^*$. This implies that there exist $i_1,\ i_2,\ i_3,\ i_4 \in \mathbf{N}$, $i_2 + i_3 \geq 1$ such that:

$$u = a^{i_1},\ v = a^{i_2},\ w = a^{N-i_1-i_2}\,b^N\,a^N\,b^{N-i_3-i_4},\ x = b^{i_3},\ y = b^{i_4}.$$

Choosing, for instance, $i = 0$, we obtain that $u\,w\,y \in L$, i.e. $a^{N-i_2}\,b^N\,a^N\,b^{N-i_3} \in L$. Since $i_2 + i_3 \geq 1$, we get neither $N - i_2 \neq N$, nor $N - i_3 \neq N$. Therefore $a^{N-i_2}\,b^N\,a^N\,b^{N-i_3}$ cannot belong to L.

(iv) Let $G = (\{S, A, B\}, \{a, b, c, d, e, f\}, S, \{S \to A \mid B,\ A \to a\,A\,b \mid c,\ B \to e\,B\,f\,f \mid d\})$ be a $LLin(1,0)$ grammar. The language generated by is $L(G) = \{a^k\,c\,b^k,\ d^k\,e\,f^{2k} \mid k \geq 1\}$. Consider the literal homomorphism defined by $h(a) = a$, $h(b) = b$, $h(c) = c$, $h(d) = a$, $h(e) = d$, $h(f) = b$. This implies that $h(L(G))$ is the language used in the proof of Theorem 3.1.3. Therefore $\mathcal{LL}in(m,n)$ is not closed under homomorphism.

■

3.2 A bidirectional parser for $LLin(m,n)$ grammars

In this section, a characterization theorem for $LLin(m,n)$ grammars and a bidirectional parser for them will be presented.

Definition 3.2.1 *Let $G = (V_N, V_T, S, P)$ be a linear grammar, $\alpha \in V^*$, $\#$ a new terminal symbol and $m, n \in \mathbf{N}^+$. We define $first_m \_ last_n(\alpha)$ as the union of the following sets of pairs of words corresponding to α, m, n so that:*

- (u, v) *if* $\exists\ \alpha \overset{*}{\underset{G}{\Longrightarrow}} u\,x\,v,\ u, x, v \in V_T^*,\ |u| = m,\ |v| = n$;
- $(x\,v\,\#, v)$ *if* $\exists\ \alpha \overset{*}{\underset{G}{\Longrightarrow}} x\,v,\ x, v \in V_T^*,\ |x\,v| = k < m,\ k \geq n,\ |v| = n$;
- $(u, \#\,u\,x)$ *if* $\exists\ \alpha \overset{*}{\underset{G}{\Longrightarrow}} u\,x,\ u, x \in V_T^*,\ |u\,x| = k < n,\ k \geq m,\ |u| = m$;

- $(x\,\#, \#\,x)$ *if* $\exists\ \alpha \overset{*}{\underset{G}{\Longrightarrow}} x,\ x \in V_T^*,\ |x| = k,\ k < m,\ k < n.$

Theorem 3.2.1 *(characterization of $LLin(m,n)$ grammars)*
Let $G = (V_N, V_T, S, P)$ be a reduced linear grammar. Then G is $LLin(m,n)$ grammar iff the following condition holds:

(1) $first_m\_last_n(\beta_1) \cap first_m\_last_n(\beta_2) = \emptyset,\ \forall A \to \beta_1,\ A \to \beta_2 \in P,\ \beta_1 \neq \beta_2.$

Proof
($\Longrightarrow$) Let us suppose that G does not satisfy condition (1). This means that there exist two distinct productions $A \to \beta_1,\ A \to \beta_2$ such that the following relation holds:

$$first_m\_last_n(\beta_1) \cap first_m\_last_n(\beta_2) \neq \emptyset.$$

According to the Definition 3.2.1, there exist four cases (# is a new terminal symbol):

1) $(u', v') \in first_m\_last_n(\beta_1) \cap first_m\_last_n(\beta_2)$. Then there exist the derivations ($|u'| = m,\ |v'| = n$):

$$\beta_1 \overset{*}{\underset{G}{\Longrightarrow}} u'\,x\,v',\ x \in V_T^*,$$

$$\beta_2 \overset{*}{\underset{G}{\Longrightarrow}} u'\,y\,v',\ y \in V_T^*.$$

Because G is a reduced grammar, it follows that A is an accessible nonterminal, so we obtain the derivations:

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_1\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,u'\,x\,v'\,v$$

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_2\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,u'\,y\,v'\,v$$

According to Definition 3.1.1, it follows that $\beta_1 = \beta_2$. *Contradiction* !

2) $(x\,v'\,\#, v') \in first_m\_last_n(\beta_1) \cap first_m\_last_n(\beta_2)$. Then according to Definition 3.2.1, there exist the derivations ($|x\,v'| = k < m,\ |v'| = n \leq k$):

$$\beta_1 \overset{*}{\underset{G}{\Longrightarrow}} x\,v',\ x \in V_T^*,$$

$$\beta_2 \overset{*}{\underset{G}{\Longrightarrow}} x\,v'.$$

So, we obtain again the derivations:

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_1\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,x\,v'\,v$$

$$S \overset{*}{\underset{G}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_2\,v \overset{*}{\underset{G}{\Longrightarrow}} u\,x\,v'\,v$$

According to Definition 3.1.1, it follows that $\beta_1 = \beta_2$. *Contradiction* !

The remaining two cases can be treated in an similar way.

($\Longleftarrow$) Let us suppose that G is not a $LLin(m,n)$ grammar. Then there exist two distinct derivations:

$$S \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_1\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,x\,v$$

$$S \underset{G}{\overset{*}{\Longrightarrow}} u\,A\,v \underset{G}{\Longrightarrow} u\,\beta_2\,v \underset{G}{\overset{*}{\Longrightarrow}} u\,y\,v$$

such as $x^{(n)} = y^{(n)}$ and ${}^{(m)}x = {}^{(m)}y$. Then there exist u', $v' \in V_T^*$ such as $|u'| = m$, $|v'| = n$ and $x = u'\,z_1\,v'$, $y = u'\,z_2\,v'$. This implies that the pair $(u', v') \in first_m\_last_n(\beta_1) \cap first_m\_last_n(\beta_2)$. But $A \to \beta_1$ and $A \to \beta_2$ are distinct productions (i.e. $\beta_1 \neq \beta_2$) in G, so we obtain a contradiction (G satisfies the condition (1)). ■

Corollary 3.2.1 *Given a linear grammar $G = (V_N, V_T, S, P)$ and two integers m and n, one can decide if the grammar is $LLin(m,n)$.*

Proof Directly from Theorem 3.2.1 and because the sets $first_m\_last_n$ can be computed (with an algorithm). ■

In the following, we shall define a device similar with a deterministic push-down "transducer". This will be called a **bidirectional parser** (syntactic analyzer, Figure 3.2) and it will be attached to a $LLin(m,n)$ grammar G. It scans an "input string", one or/and two strings at a time, from left to right or right to left. It can push or pop strings in the double ended queue (*deque*) from both sides. In the output tape, it provides the result of the syntactic analysis. It returns the values "ACC" or "ERR" depending on whether the input string is accepted or not.

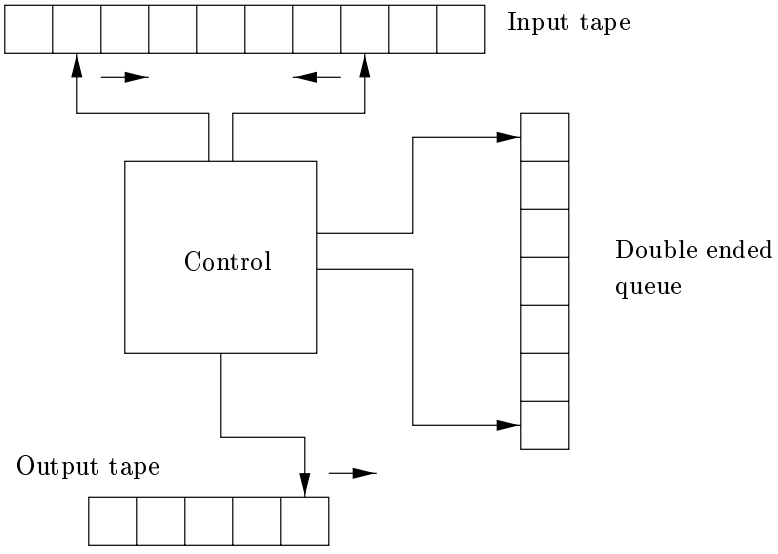


Figure 3.2. LLin(m,n) style bidirectional parser

Definition 3.2.2 *Let $G = (V_N, V_T, S, P)$ be a $LLin(m,n)$ grammar. We denote by $\mathcal{C} \subseteq \#V_T^*\# \times V^* \times \{1,2,...,|P|\}^*$ the* **set of possible configurations**, *where # is a special character (a new terminal symbol). The* **bidirectional parser** *(denoted by $BP_{m,n}(G)$) consists of the pair $(\mathcal{C}_0, \vdash)$, where the set $\mathcal{C}_0 = \{(\# w \#, S, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**, *and $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ is the* **transition binary relation** *(sometimes denoted by $\underset{BP_{m,n}(G)}{\vdash\!\!\!\!\!-\!\!\!\!\!-}$) between configurations is given by:*

1^0. **Expand transition:**

$(\#u\#, A, \lambda) \vdash (\#u\#, \beta, \pi r)$ *if* $r = no(A \to \beta)$ *for which the pair* $({}^{(m)}u\#, \#u^{(n)}) \in first_m\_last_n(\beta)$

2^0. **Reduce transitions:**

a) $(\#v_1\, u\#, v_1\, A, \pi) \vdash (\#u\#, A, \pi),\ \forall\ v_1 \in V_T^+$

b) $(\#u\, v_2\#, A\, v_2, \pi) \vdash (\#u\#, A, \pi),\ \forall\ v_2 \in V_T^+$

c) $(\#v_1\, u\, v_2\#, v_1\, A\, v_2, \pi) \vdash (\#u\#, A, \pi),\ \forall v_1, \forall v_2 \in V_T^+$

3^0. **Acceptance transition:**

$(\#\#, \lambda, \pi) \vdash ACC$

4^0. **Rejection transition:**

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

We denote by $\overset{+}{\vdash}$ ($\overset{*}{\vdash}$) the transitive (reflexive) closure of the above binary relation $\vdash$. Sometimes, for a given grammar G, we will denote these closures by $\underset{BP_{m,n}(G)}{\overset{+}{\vdash\!\!\!\!\!-\!\!\!\!\!-}}$ ($\underset{BP_{m,n}(G)}{\overset{*}{\vdash\!\!\!\!\!-\!\!\!\!\!-}}$ respectively).

It is obvious that the bidirectional parser $BP_{m,n}(G)$ is deterministic, i.e. for any arbitrary configuration at most one configuration may be "reached". But the condition $({}^{(m)}(u\#), (\#u)^{(n)}) \in first_m\_last_n(\beta)$ ensures the uniqueness of the production $A \to \beta$ because G is a $LLin(m,n)$ grammar.

Lemma 3.2.1 *Let G be a $LLin(m,n)$ grammar. Then, the following implications are true:*

(i) if $(\#v_1\, u\, v_2\#, S, \lambda) \underset{BP_{m,n}(G)}{\overset{*}{\vdash\!\!\!\!\!-\!\!\!\!\!-}} (\#u\#, X, \pi')$ *then* $S \underset{G}{\overset{\pi'}{\Longrightarrow}} v_1\, X\, v_2$;

(ii) if $(\#w\#, S, \lambda) \underset{BP_{m,n}(G)}{\overset{*}{\vdash\!\!\!\!\!-\!\!\!\!\!-}} (\#\#, \lambda, \pi)$ *then* $S \underset{G}{\overset{\pi}{\Longrightarrow}} w$.

Proof

(i) By induction on the length of π'.

Basis: $|\pi'| = 0$. Thus $v_1 = v_2 = \lambda,\ A = S$, and then $S \underset{G}{\overset{\lambda}{\Longrightarrow}} S$.

Inductive Step: Let $\pi' = \pi'_1\ r$, where $r = no(B \to \beta)$ is the associated number of the last applied production. Denoting $v_1 = v_{11}\, v_{12}$ and $v_2 = v_{21}\, v_{22}$ we obtain:

$$(\#v_1\, u\, v_2\#, S, \lambda) = (\#v_{11}\, v_{12}\, u\, v_{21}\, v_{22}\#, S, \lambda) \overset{*}{\vdash} (\#v_{12}\, u\, v_{21}\#, B, \pi'_1).$$

From the inductive hypothesis, it follows that $S \underset{G}{\overset{\pi'_1}{\Longrightarrow}} v_{11}\, B\, v_{22}$. Then from 1^0 (Definition 3.2.2), we obtain the configuration $(\#v_{12}\, u\, v_{21}\#, \beta, \pi_1\, r)$, where $({}^{(m)}v_{12}\, u\, v_{21}\, \#, \#\, v_{12}\, u\, v_{21}^{(n)}) \in first_m\_last_n(\beta)$. The next transitions

$$(\#v_{12}\, u\, v_{21}\#, \beta, \pi'_1\, r) \underset{BP_{m,n}(G)}{\overset{*}{\vdash}} (\#u\#, X, \pi')$$

could be only reduce transitions. So $\beta = v_{12}\, X\, v_{21}$ (2^0 a),b),c), Definition 3.2.2). We may then have the derivation:

$$S \underset{G}{\overset{\pi'_1}{\Longrightarrow}} v_{11}\, B\, v_{22} \underset{G}{\overset{r}{\Longrightarrow}} v_{11}\, \beta\, v_{22} = v_{11}\, v_{12}\, X\, v_{21}\, v_{22} = v_1\, X\, v_2$$

(ii) Take $u = \lambda,\ X = \lambda,\ v_1\, v_2 = w,\ \pi' = \pi$ in (i) . ■

Lemma 3.2.2 *Let G be a $LLin(m,n)$ grammar. Then, the following implications hold:*

(i) $S \underset{G}{\overset{\pi'}{\Longrightarrow}} v_1\, X\, v_2$ *implies* $(\#v_1\, u\, v_2\#, S, \lambda) \underset{BP_{m,n}(G)}{\overset{*}{\longmapsto}} (\#u\#, X, \pi')$;

(ii) $S \underset{G}{\overset{\pi}{\Longrightarrow}} w$ *implies* $(\#w\#, S, \lambda) \underset{BP_{m,n}(G)}{\overset{*}{\longmapsto}} (\#\#, \lambda, \pi)$;

Proof

(i) By induction on the length of π'.

Basis: $|\pi'| = 0$. Thus $v_1 = v_2 = \lambda,\ A = S$, and following transitions hold:

$$(\#u\#, S, \lambda) \underset{BP_{m,n}(G)}{\overset{*}{\vdash}} (\#u\#, S, \lambda).$$

Inductive Step: Let $\pi' = \pi'_1\, r$, where $r = no(B \to \beta)$ be associated number of the last applied production which generates the sentential form $v_1\, X\, v_2$. The above derivation may be written as:

$$S \underset{G}{\overset{\pi'_1}{\Longrightarrow}} v_{11}\, B\, v_{22} \underset{G}{\overset{r}{\Longrightarrow}} v_{11}\, v_{12}\, X\, v_{21}\, v_{22} = v_1\, X\, v_2$$

Applying the inductive hypothesis for π_1', we obtain

$$(\#v_{11}\, v_{12}\, u\, v_{21}\, v_{22}\#, S, \lambda) \overset{*}{\underset{BP_{m,n}(G)}{\vdash}} (\#v_{12}\, u\, v_{21}\#, B, \pi_1').$$

Now, we may continue with expand transition, obtaining the configuration $(\#v_{12}\, u\, v_{21}\#, v_{12}\, X\, v_{21}, \pi_1'\, r)$. Then we apply the reduce transitions a),b),c) and obtain the configuration $(\#u\#, X, \pi_1'\, r) = (\#u\#, X, \pi')$.

(ii) Take $u = \lambda$, $X = \lambda$, $v_1\, v_2 = w$, $\pi' = \pi$ in (i). ■

Theorem 3.2.2 *(correctness and complexity of $BP_{m,n}(G)$)*
Let G be a $LLin(m,n)$ grammar. Then

$$(\#w\#, S, \lambda) \overset{*}{\underset{BP_{m,n}(G)}{\vdash}} (\#\#, \lambda, \pi) \underset{BP_{m,n}(G)}{\vdash} ACC \text{ iff } S \overset{\pi}{\underset{G}{\Longrightarrow}} w.$$

On the other hand, $(\#w\#, S, \lambda) \overset{*}{\underset{BP_{m,n}(G)}{\vdash}} ERR$ iff $w \notin L(G)$. The number of transitions of $BP_{m,n}(G)$ is $k \cdot |w|$, where w is the input word and k is a positive integer constant.

Proof Both implications directly follow from Lemmas 3.2.1 (ii) and 3.2.2 (ii), respectively. The time complexity results from the fact that $BP_{m,n}(G)$ is defined over a finite structure (grammar G) and $BP_{m,n}(G)$ is deterministic (i.e. for any given configuration, at most one transition could be applied). ■

The next section will be dedicated to another practical bidirectional parser, for $LLin(1,1)$ grammars (the sets $first_1\_last_1$ can be computed in polynomial time related to the dimension of the input grammar).

3.3 Bidirectional parsing for $LLin(1,1)$ grammars

The $LLin(0,0)$ grammars have the property that there exists no nonterminal symbol which may be in the left hand side of a production. Obviously, for a reduced $LLin(0,0)$ grammar, its language is finite.

We also do not consider $LLin(1,0)$ or $LLin(0,1)$ grammars because they coincide with $LLin(1)$ grammars or reverse (mirror) $LLin(1)$ grammars (Corollary 3.1.1).

Definition 3.3.1 *Let $G = (V_N, V_T, S, P)$ be a linear grammar, $\alpha \in V^*$. Then*
$first\_last(\alpha) = \{(a,b) \mid \exists \alpha \overset{*}{\underset{G}{\Longrightarrow}} a\, v\, b,\ v \in V_T^*,\ a,\ b \in V_T\} \cup$
$\cup \{(a,a) \mid \exists \alpha \overset{*}{\underset{G}{\Longrightarrow}} a,\ a \in V_T\} \cup \{(\#,\#) \mid \alpha \overset{*}{\underset{G}{\Longrightarrow}} \lambda\}$

Theorem 3.2.1 becomes:

Theorem 3.3.1 *G is $LLin(1,1)$ grammar iff $first\_last(\beta_1) \cap first\_last(\beta_2) = \emptyset$, $\forall\ A \to \beta_1 \in P$, $\forall\ A \to \beta_2 \in P$, $\beta_1 \neq \beta_2$.*

The bidirectional parser $BP_{1,1}(G)$ (denoted simply by $BP(G)$) can also be reformulated (we present only the transition relation, # being a new nonterminal symbol):

1^0 **Expand transition:**

$(\#u\#, A, \pi) \vdash (\#u\#, \beta, \pi\, r)$ if $r = no(A \to \beta)$ and the pair $({}^{(1)}u\#, \#u^{(1)}) \in first\_last(\beta)$

2^0. **Reduce transitions:**

a) $(\#v_1\, u\#, v_1\, A, \pi) \vdash (\#u\#, A, \pi),\ v_1 \in V_T^+$

b) $(\#u\, v_2\#, A\, v_2, \pi) \vdash (\#u\#, A, \pi),\ v_2 \in V_T^+$

c) $(\#v_1\, u\, v_2\#, v_1\, A\, v_2, \pi) \vdash (\#u\#, A, \pi),\ v_1,\ v_2 \in V_T^+$

3^0. **Acceptance transition:**

$(\#\#, \lambda, \pi) \vdash ACC$

4^0. **Rejection transition:**

$(\#u\#, \alpha, \pi) \vdash ERR$ if no transitions of type $1^0,\ 2^0,\ 3^0$ can be applied.

$BP(G)$ may be used in practical compiler applications. For instance the computation of the sets $first\_last(\alpha)$ (α being right hand side of a production) can be done in polynomial time on the dimension of input linear grammar G.

Example 3.3.1 *Let us review the grammar G_4 from Example 3.1.1.*

1. $S \to a\, S$
2. $S \to A\, b$
3. $A \to A\, b$
4. $A \to \lambda$

We can easily "compute" the sets:

- $first\_last(a\, S) = \{(a, b)\}$;
- $first\_last(A\, b) = \{(b, b)\}$;
- $first\_last(\lambda) = \{(\#, \#)\}$;

According to Theorem 3.3.1, it follows that G_4 is a $LLin(1,1)$ grammar. Let us now consider the word $w = a\,a\,b\,b\,b$. In $BP(G_4)$ we have:

$$(\#a\,a\,b\,b\,b\#, S, \lambda) \vdash (\#a\,a\,b\,b\,b\#, a\,S, [1]) \vdash (\#a\,b\,b\,b\#, S, [1]) \vdash$$

$$\vdash (\#a\,b\,b\,b\#, a\,S, [1,1]) \vdash (\#b\,b\,b\#, S, [1,1]) \vdash (\#b\,b\,b\#, A\,b, [1,1,2]) \vdash$$

$$\vdash (\#b\,b\#, A, [1,1,2]) \vdash (\#b\,b\#, A\,b, [1,1,2,3]) \vdash (\#b\#, A, [1,1,2,3]) \vdash$$

$$(\#b\#, A\,b, [1,1,2,3,3]) \vdash (\#\#, A, [1,1,2,3,3]) \vdash (\#\#, \lambda, [1,1,2,3,3,4]) \vdash ACC$$

So, w is "accepted" by $BP(G_4)$. According to Theorem 3.2.2, it follows that $w \in L(G_4)$.

Two additional functions and two additional binary relations are needed for determining the sets $first\_last(\alpha)$, where α is a right hand side of a production of G.

These are $\mathtt{first}$, $\mathtt{last} : V_N \to \mathcal{P}(V_T) \cup \{\lambda\}$ and $\mathtt{begin}$, $\mathtt{end} \subseteq V \times V_N$ given by:

- $a \in \mathtt{first}(A)$ iff there exists a derivation $A \overset{*}{\underset{G}{\Longrightarrow}} a\,\alpha$;
- $a \in \mathtt{last}(A)$ iff there exists a derivation $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha\,a$;
- $\lambda \in \mathtt{first}(A)$ (or $\mathtt{last}(A)$) iff there exists a derivation $A \overset{*}{\underset{G}{\Longrightarrow}} \lambda$;
- $X\ \mathtt{begin}\ A$ iff there exist a production $A \to \beta\,X\,v$ and a production $\beta \overset{*}{\underset{G}{\Longrightarrow}} \lambda$, where $\beta \in V_N \cup \{\lambda\}$;
- $X\ \mathtt{end}\ A$ iff there exist a production $A \to u\,X\,\beta$ and a production $\beta \overset{*}{\underset{G}{\Longrightarrow}} \lambda$, where $\beta \in V_N \cup \{\lambda\}$.

The following lemma suggests a procedure for obtaining the relations $\mathtt{begin}$ and $\mathtt{end}$.

Lemma 3.3.1

1) If $Y\ \mathtt{begin}^n\ X$ then there exists m, $m \geq n$ such that $X \overset{m}{\underset{G}{\Longrightarrow}} Y\,\alpha$;

2) If $X \overset{n}{\underset{G}{\Longrightarrow}} Y\,\alpha$ then there exists m, $m \leq n$ such that $Y\ \mathtt{begin}^m\ X$;

3) $a\,\mathtt{begin}^*\ A$ iff there exists a derivation $A \overset{*}{\underset{G}{\Longrightarrow}} a\,\alpha$;

4) If $Y\ \mathtt{end}^n\ X$ then there exists m, $m \geq n$ such that $X \overset{m}{\underset{G}{\Longrightarrow}} \alpha\,Y$;

5) If $X \overset{n}{\underset{G}{\Longrightarrow}} \alpha\,Y$ then there exists m, $m \leq n$ such that $Y\ \mathtt{end}^m\ X$;

6) $a\,\mathtt{end}^*\,A$ *iff there exists a derivation* $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha\,a$.

Proof By induction on m and n. ■

According to Lemma 3.3.1, it is obvious that:

- $a \in \mathtt{first}(A)$ iff $a\,\mathtt{begin}^*\,A$;
- $a \in \mathtt{last}(A)$ iff $a\,\mathtt{end}^*\,A$.

The computation of $first\_last$ may be given as the value returned of the following self-explanatory recursive function.

Input: The linear grammar $G = (V_N, V_T, S, P)$
Output: $first\_last(\alpha),\ \alpha \in V^*$.

function $first\_last(\alpha)$;
begin
  if $(\alpha = \lambda)$ **then** $first\_last(\alpha) := \{(\#, \#)\}$;
  if $(\alpha = a,\ a \in V_T)$ **then** $first\_last(\alpha) := \{(a, a)\}$;
  if $(\alpha = a\,\beta\,b,\ a, b \in V_T)$ **then** $first\_last(\alpha) := \{(a, b)\}$;
  if $(\alpha = A\,u\,b,\ A \in V_N,\ u \in V_T^*,\ b \in V_T)$ **then begin**
    $first\_last(\alpha) := \{(a, b) \mid a \in \mathtt{first}(A) - \{\lambda\}\}$;
    if $(\lambda \in \mathtt{first}(A))$ **then** add to $first\_last(\alpha)$ the pair $({}^{(1)}u\,b, b)$;
  end ;
  if $(\alpha = a\,u\,A,\ a \in V_T,\ u \in V_T^*,\ A \in V_N)$ **then begin**
    $first\_last(\alpha) := \{(a, b) \mid a \in \mathtt{last}(A) - \{\lambda\}\}$;
    if $(\lambda \in \mathtt{last}(A))$ **then** add to $first\_last(\alpha)$ the pair $(a, a\,u^{(1)})$;
  end ;
  if $(\alpha = A,\ A \in V_N)$ **then begin**
    $set\_chain(A) := \{B \mid A \overset{*}{\underset{G}{\Longrightarrow}} B,\ B \in V_N\}$;
    $set\_fst\_snd := \emptyset$;
    for (any $A \to \beta \in P,\ B \in set\_chain(A)$) **do**
      if $(\beta \notin V_N)$ **then** $set\_fst\_snd := set\_fst\_snd \cup first\_last(\beta)$;
    $first\_last(\alpha) := set\_fst\_snd$
  end
end .

The previous algorithm (for computing the function $first\_last$) has polynomial time complexity (on the dimension of G) because it describes (in a recursive manner) the transitive closure of the derivation relation for linear grammars.

The following example proves that $first\_last(\alpha)$ is properly included in $first(\alpha) \times last(\alpha)$.

Example 3.3.2 *Let* $G = (\{S, A\}, \{a, b, c\}, S, \{S \to A,\ A \to a\,A\,b \mid b\,A\,a \mid c\})$ *be a linear grammar. we have* $first\_last(A) = \{(a, b), (b, a), (c, c)\}$. *On the other hand,* $\mathtt{first}(A) = \{a, b, c\}$ *and* $\mathtt{last}(A) = \{a, b, c\}$. *It results that* G *is a* $LLin(1, 0)$ *(or* $LLin(0, 1)$*) grammar.*

3.4 Conclusions

Following the stated results related to $LLin(m,n)$ grammars, the following "inclusion" diagram holds:

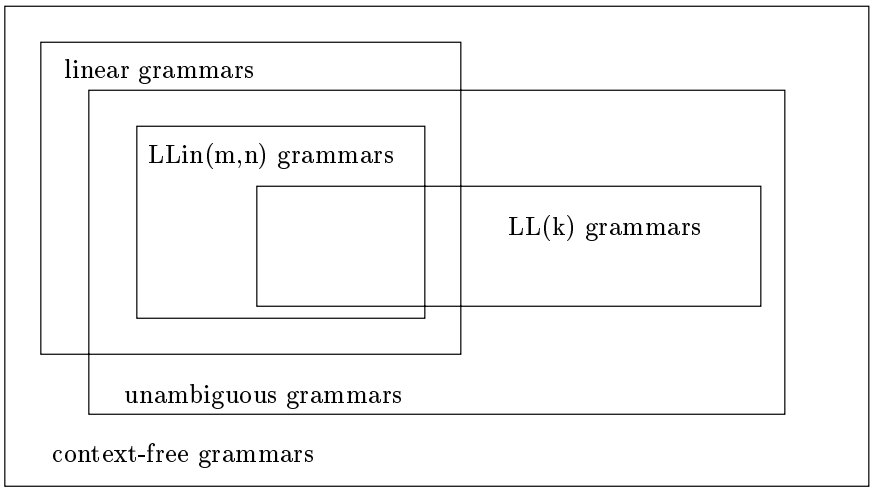


Figure 3.3

Without loss of generality, we allow - from now on - three modifications of the bidirectional parser for testing the power of the device given in Definition 3.2.2:

(i) reading (and/or replacing) of two consecutive symbols (at the ends of the deque);

(ii) interchanging the contents of the mentioned two ends of the deque;

(iii) removing the third component, i.e. the syntactic analysis.

Accordingly, we can now present an example of a bidirectional parser which can analyze 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:
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 the initial configuration, we take $(\#w\#, A)$, where $w \in \{a,b,c\}^*$ is the input word. Assuming that the notations w and γ stand for words (of any length) over $\{a,b,c\}$, and $\{a,b,c,A,B\}$ respectively, the transitions will be the following:

1. $(\#a\, a\, w\, c\#, A\, \gamma) \vdash (\#a\, a\, w\, c\#, a\, A\, B\, c\, \gamma)$
2. $(\#a\, w\, c\#, a\, \gamma\, c) \vdash (\#w\#, \gamma)$
3. $(\#a\, w\#, a\, \gamma\, B) \vdash (\#w\#, \gamma\, B)$
4. $(\#a\, b\, w\#, A\, \gamma) \vdash (\#a\, b\, w\#, a\, b\, c\, \gamma)$

5. $(\#b\,w\,c\#, b\,\gamma\,B) \vdash (\#w\,c\#, \gamma\,B)$
6. $(\#b\,w\,c\#, c\,\gamma\,c\,B) \vdash (\#b\,w\,c\#, c\,\gamma\,B\,c)$
7. $(\#b\,w\,c\#, c\,B\,\gamma\,c) \vdash (\#b\,w\,c\#, B\,c\,\gamma\,c)$
8. $(\#b\,w\#, B\,\gamma) \vdash (\#b\,w\#, b\,\gamma)$
9. $(\#b\,w\,c\#, b\,\gamma\,c) \vdash (\#w\#, \gamma)$
10. $(\#b\,b\,w\,c\,c\#, c\,c\,\gamma\,B\,B) \vdash (\#b\,b\,w\,c\,c\#, B\,c\,\gamma\,B\,c)$
11. $(\#b\,c\#, c\,B) \vdash (\#b\,c\#, B\,c)$
12. $(\#\#, \lambda) \vdash ACC$
13. $(\cdot, \cdot) \vdash ERR$ - in the other cases.

The above bidirectional parser is deterministic because at each step at most one transition may be applied. We may say that the parser is of type $(3, 3)$ because at the transition 11, we need to read three symbols from the left, and right, respectively.

We conclude that the subclasses of $\mathcal{LL}in(m, n)$ languages are more "powerful" than some deterministic context-free languages, "keeping" the linear time complexity of the associated algorithms for solving the membership problem. Known closure properties are generally not preserved.

Open-problems: Are the $\mathcal{LL}in(m, n)$ languages closed under complementation, intersection with regular languages and inverse homomorphism ?

Chapter 4

Left and right bidirectional parsing for context free grammars

In this chapter we describe some subclasses of context-free grammars for which a parallel approach useful for solving the membership problem will be defined ([AnK99a]). More precisely, we will combine the classical type of parser attached to a grammar G with a "mirror" process for G. They simultaneously analyze the input word from both sides, using - ideally - 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 - mainly - contains two algorithms which use a backtracking method to describe the nondeterministic behavior 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 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 types of grammars can be solved with a parallel algorithm in linear time.*

4.1 Left and right bidirectional parsers

We define two parsers which have the main goal to accept context free languages. The idea is similar - but more general - to that of [AnK98]. The input word

is still analyzed from both sides, but the given algorithm "works" (having two "heads" which operate independently) not only for the class of linear grammars, but for context free grammars.

Before giving the description of a new kind of parsers for context free grammars, a list of specific definitions and notations are in order.

Firstly a device similar to a nondeterministic pushdown "transducer" is needed. This will be called the **general bidirectional parser attached to the context free grammar** G**.** It scans two "input 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 rejected.

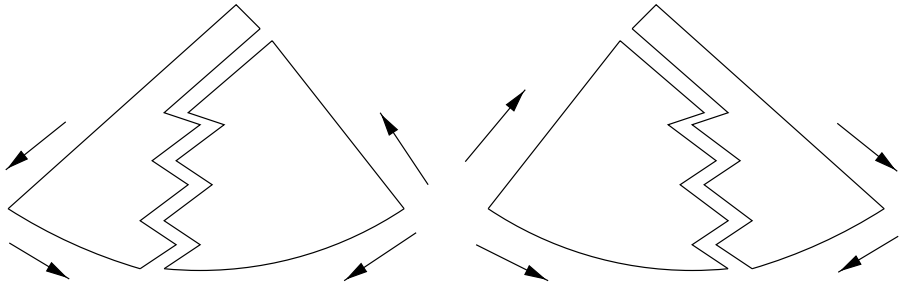
In Section 4.4, we shall "see" how a deterministic bidirectional parser can be designed (for some particular subclasses of context free languages).

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

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

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

These two strategies can be depicted as:



(a) (b)

Figure 4.1. Strategies for left and right bidirectional parsing

We shall present in the following - in a formal manner - only the situation (a).

Definition 4.1.1 *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 new special character (a 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 a state, the second and the last components of a configuration store the partial syntactic analysis. The third and the fifth components represent the working stacks (each of which having at the bottom the marker #). The fourth component represents the current content of the input word (enclosed by the two markers). The* **transition relation** *($\vdash \subseteq \mathcal{C} \times \mathcal{C}$, sometimes denoted by $\underset{GBP_l(G)}{\vdash}$) 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* $r = no(A \to \delta)$;

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* $r_1 = no(A \to \delta)$ *and* $r_2 = no(B \to \varepsilon)$;

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* $r = no(B \to \varepsilon)$;

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* $r = no(A \to \delta)$;

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* $r = no(B \to \varepsilon)$;

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 tape 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. In [HoU79], there exists Lemma 7.3:

An arbitrary single-tape Turing machine can be simulated by a deterministic two-stack machine.

Another model equivalent to Turing machine is the two-counter machine, which is off-line Turing machine whose storage is semi-infinite (e.g. on the right), and whose alphabets contain only two symbols, Z and B (blank). Furthermore,

the symbol Z, which denoted the bottom of the stack, occurs initial 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. A two-counter machine can test whether a number is zero by checking whether Z is scanned by the head, but it cannot directly test whether two numbers are equal. In [HoU79] there exists Theorem 7.9:

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 of two output tapes which can be accessed only in write style, and has only two states. According to the results presented above, our model can simulate a Turing machine.

This model can be depicted as:

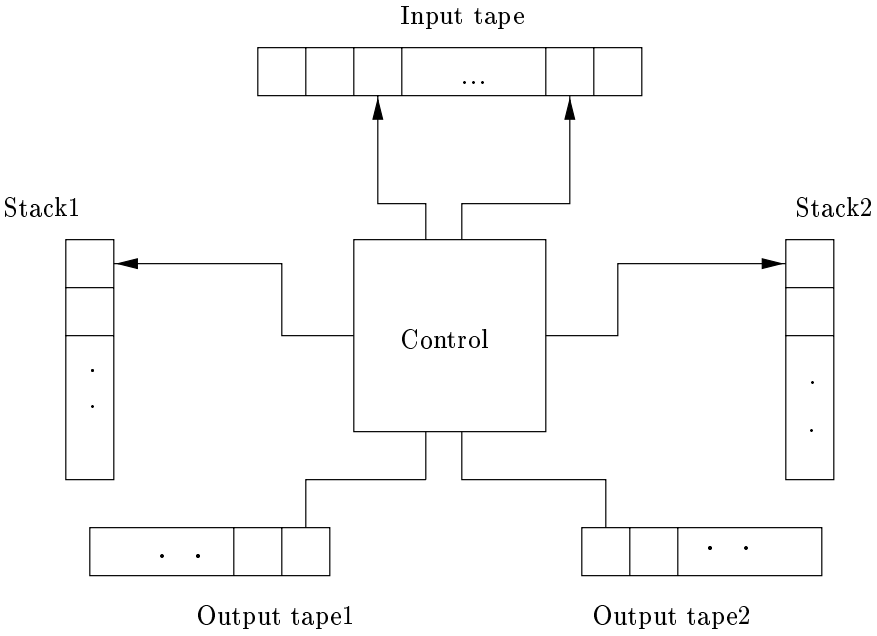


Figure 4.2. General Left and Right Bidirectional Parser Style

Lemma 4.1.1 *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 applied transitions (de-

noted by t) during the computation (1). $\overset{t,*}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}}$ and $\overset{1^0}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}}$ will mean that t transitions, respectively that transition number 1^0 have been applied.
Basis: $t = 1$. Starting from the initial configuration, we can successively apply the transitions 1^0, 5^0 and 7^0. Supposing that we apply 1^0. Then we obtain:

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

where $r = S \to \alpha \in P$ and $u_3 = u_3'\, a$. Therefore $S \underset{G,lm}{\overset{r}{\Longrightarrow}} \alpha$ and $a \underset{G,lm}{\overset{0}{\Longrightarrow}} a$. The remaining cases ($5^0$, 7^0) can be treated in a similar way.
Inductive Step: Suppose that relation (1) is true for at most t transitions and prove it for $t+1$ (applied) transitions. We know that:

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

We have to prove that $S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1\, \alpha$ and $\beta \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} u_3$. The last transition in (2) may be of one of the types 1^0, 2^0, ..., 8^0.
I: Suppose that the last transition in (2) is *expand-shift*. We may rewrite (2) as:
$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \overset{t,*}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1', A\alpha_2\#, \#u_2\, b\#, \beta'\#, \pi_2)$
$\overset{1^0}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$, where $r = no(A \to \alpha_1)$, $\pi' r = \pi_1$,
$\alpha_1\, \alpha_2 = \alpha$, $b\, \beta' = \beta$. According to the inductive hypothesis, we obtain:

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

But $A \to \alpha_1$ is the $r-$th production from P, and so:

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

II: Suppose that the last transition in (2) is *expand-reduce*. We can rewrite the initial transition as:
$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \overset{t,*}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1', A\, \alpha_2\#, \#u_2\#, \varepsilon\, \beta'\#, \pi_2)$
$\overset{2^0}{\underset{GBP_l(G)}{\vdash\!\!\!-\!\!\!-}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $r_1 = no(A \to \alpha_1)$, $\pi_1' r_1 = \pi_1$,
$\alpha_1\, \alpha_2 = \alpha$, $r_2 = no(B \to \varepsilon)$, $B\, \beta' = \beta$, $r_2 \pi_2' = \pi_2$. According again to the inductive hypothesis, we get:

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

But $A \to \alpha_1$ and $B \to \varepsilon$ are the r_1-th, respectively r_2-th, productions from P, and then

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

III: Suppose that the last transition in (2) is *reduce-shift*. The initial transition may be rewritten as:

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

$\underset{GBP_l(G)}{\overset{3^0}{\vdash}} (\pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $b\,\beta' = \beta$. Following 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: Suppose that the last transition in (2) is *reduce-reduce*. The initial transitions become:

$(s_1, \lambda, S\#, \#u_1\, u_2\, u_3\#, \#, \lambda) \underset{GBP_l(G)}{\overset{t,*}{\vdash}} (s_1, \pi_1, a\, \alpha\#, \#a\, u_2\#, \delta\, \beta'\#, \pi_2')$

$\underset{GBP_l(G)}{\overset{4^0}{\vdash}} (s_1, \pi_1, \alpha\#, \#u_2\#, \beta\#, \pi_2)$ where $r = no(B \to \delta)$, $B\,\beta' = \beta$, $r\pi_2' = \pi_2$.

According to the inductive hypothesis, we get:

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

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*) may be similarly treated. ■

Lemma 4.1.2 *Let G be a context free grammar. If $S \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} u_1\, \alpha$ and $\beta \underset{G,lm}{\overset{\pi_2}{\Longrightarrow}} u_3$, where ${}^{(1)}\alpha \in V_N$ or $\alpha = \lambda$, then*

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

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

Basis:

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

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

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

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

  Two cases may distinguished:

  - $|u_1| \geq |u_3|$. Then the configuration may be continued with a number of $|u_1|$ transitions of the form *reduce* - *shift* (finally) obtaining the configuration:

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

    Now, we may apply $|u_3'|$ transitions of the form *stay* - *shift* to get the final configuration:

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

  - $|u_1| < |u_3|$. We may continue with $|u_3|$ transitions of the form *reduce* - *shift* to obtain 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:

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

- $t = 1$. We suppose $|\pi_1| = 0$ 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 a number of $|u_3|$ transitions of the type *stay* - *shift* to obtain:

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

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

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

Inductive Step: We have to prove that $P(t) \rightarrow P(t+1)$, where P is the (obvious) logical predicate equivalent to our implication. We thus have to 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 $A \rightarrow \delta$ be the last applied production (numbered by r_1) 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 ($\alpha_1'' \, \alpha_1' = \alpha$ and $\pi_2' = \pi_2$):

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

II: Let $B \rightarrow \varepsilon$ be the last applied production (numbered by r_2) 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 4.1.1 *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) Simply take $u_1 = u_2 = \lambda$, $u_3 = w$, $\alpha = \beta = \gamma$ in Lemmas 4.1.1 and 4.1.2,
b) The unique transition for which we can obtain an ACC answer 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 $(s_1, \lambda, S\#, \#w\#, \#, \lambda) \overset{*}{\underset{GBP_l(G)}{\vdash}} (s_2, \pi_1, \gamma\#, \#\#, \gamma\#, \pi_2)$. 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 $\gamma \in V^*$ for which

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

According to a), we get

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

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

For (b) from Figure 4.1, we have just to "reverse" the general left bidirectional parser (by switching the first two components with the last two components of the given configurations).

Definition 4.1.2 *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 new special character (a terminal symbol). The* **general right bidirectional parser** *($GBP_r(G)$) is given by a 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:* $(s_1, \pi_1, \#\beta, \#b\,u\#, \#\alpha\,A, \pi_2) \vdash (s_1, \pi_1, \#\beta b, \#u\#, \#\alpha\,\delta, \pi_2 r)$, *where* $r = no(A \to \delta)$;
- 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* $r_1 = no(B \to \varepsilon)$ *and* $r_2 = no(A \to \delta)$;
- 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* $r = no(B \to \varepsilon)$;
- 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* $r = no(A \to \delta)$;

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 relation between the right general bidirectional parser associated to a grammar G and the right most derivations in G.

Lemma 4.1.3 *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 4.1.1 (using induction on the number of transitions). ■

Lemma 4.1.4 *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 4.1.2, (by induction on $|\pi_1| + |\pi_2|$). ■

Theorem 4.1.2 *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 By taking $u_1 = w$, $u_2 = u_3 = \lambda$, $\alpha = \beta = \gamma$ in Lemmas 4.1.3 and 4.1.4. ■

Example 4.1.1 *Let us consider the context free grammar given by the productions* $\mathbf{1}.A \to a\,A\,B\,b$ $\mathbf{2}.A \to c\,d$ $\mathbf{3}.B \to B\,C\,e$ $\mathbf{4}.B \to f$ $\mathbf{5}.C \to g$, *and the word* $w = a\,c\,d\,f\,g\,e\,b$. *We shall see how* $GBP_l(G)$ *and* $GBP_r(G)$ *can work on the input the word* w *(above the sign "*$\vdash$*" the associated transition number may be indicated).*

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

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

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* $S \overset{\pi_{lm}}{\underset{G}{\Longrightarrow}} w$.

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

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

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* $S \overset{\pi_{rm}}{\underset{G}{\Longrightarrow}} w$.

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" into the configuration $(s_1, [1,2], B\,\beta\#, \#\#, B\,C\,e\,b\#, [4,5])$.

The same can be "said" 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 4.3, 4.4 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 behavior for practical compilers.

Example 4.1.2 *In order to test the power of the bidirectional parser models (Definitions 4.1.1 and 4.1.2), we shall allow the simultaneously 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 analyze 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 the 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 of the following forms:*

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.*

The above bidirectional parser is deterministic because for each configuration at most one transition may be applied.

Theorem 4.1.3 *The following statement holds:*

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

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

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

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

$(c\,(B\,c)^{n-1}\#, \#b^{n-1}\#, c^n\#) \overset{5^0}{\vdash} ((B\,c)^{n-1}\#, \#b^{n-1}\#, c^{n-1}\#) \overset{6^0}{\vdash}$

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

($\Longrightarrow$) 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 $A \overset{*}{\Longrightarrow} w$.

Beginning with 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 reach the configuration $(a\,b\,c\#, \#a\,b\,u\#, \#)$. Now only 3^0, i.e. $u = u_1\,c$, may be applied. We get the configuration $(b\,c\#, \#b\,u'\#, c\#)$. Applying the transition 4^0, we obtain $(c\#, \#u'\#, c\#)$. The only possibility to obtain ACC is to apply 5^0 and get $(\#, \#u'\#, \#)$; it is obvious that $u' = \lambda$ and therefore $w = a\,b\,c$ $(n = 1)$.

If we apply 1^0, then $w = a\,a\,u$, and we obtain $(a\,A\,B\,c\#, \#a\,a\,u\#, \#)$. Again, only a type 3^0 transition is applicable, i.e. $u = u_1\,c$. We get the configuration $(A\,B\,c\#, \#a\,u_1\#, c\#)$. Only transitions 1^0 and 2^0 may now be applied. 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 transitions of type 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:

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

It follows - applying 4^0 - that $u'_1 = b\,v_1$ and the next configuration is:

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

The only possibility is - at this moment - to apply 5^0. We get:

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

Now, we can apply only 6^0. We get 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

$$w = a\,a\,u = a\,a\,u_1\,c = a\,a\,a^{n-2}\,u'\,c^{n-2}\,c = a^n\,u'_1\,c^n =$$

$$= a^n\,b\,v_1\,c^n = a^n\,b^n\,v'_1\,c^n = a^n\,b^n\,c^n\ (n \geq 2)$$

■

4.2 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 4.3).

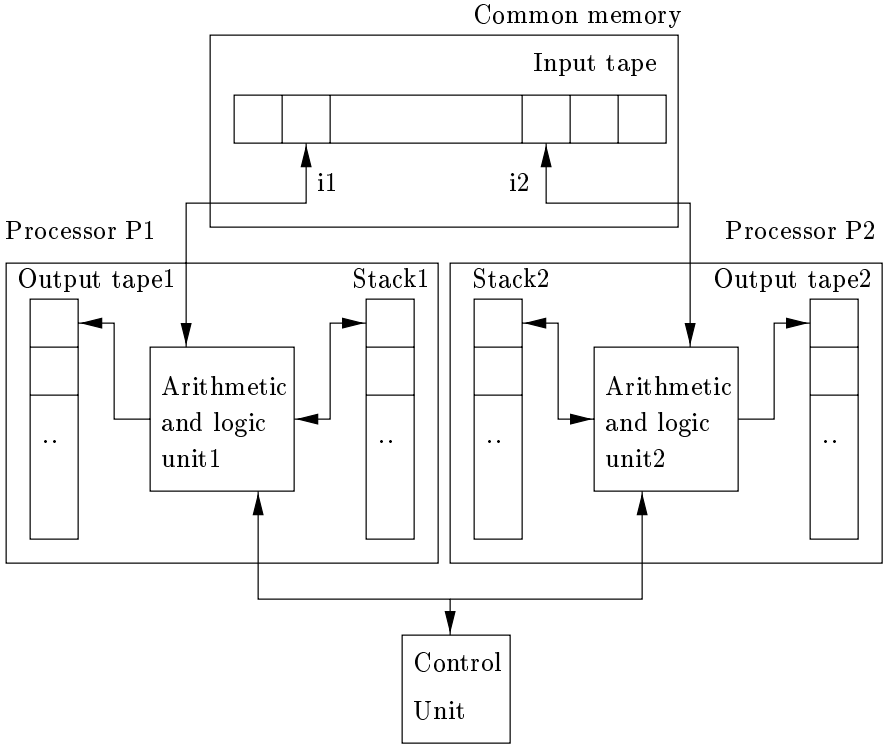


Figure 4.3. General SIMD Model for Left and Right Bidirectional Parsing

Following [Akl97], our model is a SIMD (simple instruction stream, 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. They share a common memory.

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

- `w` $\in V_T^*$ is the input word; `w` can be also stored into the local memories of the two processors; in that case, the only global variables (belonging to the common memory) are `i1` and `i2`;
- `n` is the length of `w`;
- `i1`, `i2` are two counters indicating the (current) positions of the pointers

to `w`;

- `is_no_over` is a boolean variable which takes the value `true` if processor `P2` performs a reduce action;
- `accept` is a boolean variable which takes the value `true` iff $w \in L(G)$;
- `Stack1`, `Stack2` are the two working stacks for `P1` and `P2`;
- `Output_tape1`, `Output_tape2` are the output tapes of `P1` and `P2` and will be used for storing the syntactic analysis of the input word;
- `exit` is a boolean variable which is true iff `P1` or `P2` detect the non-acceptance situation for the input word.

We shall use also some predefined procedures, such as:

- `pop(Stack,top)` - the value of `top` will be set to the value of the first symbol from `Stack`; after that, the top of `Stack` will be removed;
- `push(Stack,`α`)` - push into 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 into the top of `Output_tape` the value of `r`.

Supposing that the context free grammar $G = (V_N, V_T, S, P)$ is already given, the method of (PAR_LEFT)is listed below:

```
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 hand syntactic analysis ');
     write(Output_tape1, Output_tape2);
```

```
   end
   else write('w is not accepted.');
end;
```

It remains to list 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 top → α ∈ P, where r = no(top → α);
       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 backtracking steps ended) and
         (still no reduce or expand action could be performed)
       then exit:=true
     end
end;

procedure action2(P2);
begin
   case
     if (∃ r= no(B → β), 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 backtracking steps ended) and
```

```
          (still no reduce or expand action could be performed)
        then exit:=true
      end
end;
```

Theorem 4.2.1 *(termination of the Algorithm (PAR_LEFT))*
The Algorithm (PAR_LEFT) performs a finite number of steps until it terminates its execution.

Proof Because the input grammar G has a finite number of productions, the number of possible expand actions for `P1` and reduce actions for `P2` is finite. For the other actions (i.e. reduce for `P1` and shift for `P2`) only one character from `w` is read. Because `w` is a finite word it follows that the statement `repeat-until` from (PAR_LEFT) performs 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 4.2.2 *(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 pseudo-code description of $GBP_l(G)$ (according to Definition 4.1.1). More precisely, for showing the correctness of the parallel Algorithm (PAR_LEFT), it suffices to note 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 perform only expand actions, while P2 could perform both operations, i.e. *reduce* and *shift*. This restriction was imposed in order to eliminate the (possible) errors and the *deadlock*. Deadlock in the algorithm is understood in the following sense: both processors wait one each other to "read" the letter. Some errors come from the fact that both processors actually "read" the letter (i.e. P1 performs `i1:=i1+1`, and P2 performs `i2:=i2-1`) and thus the content of the two stacks could not be equal, even for words which belong to the language of the input grammar.

Consider now the following cases:

(i) P1 works, P2 works

(ii) P1 stays, P2 works

(iii) P1 works, P2 stays

It is obvious that situation (i) for which `action1(P1)` and `action2(P2)` are called and executed in parallel, is described in an "equivalent" way in general

left bidirectional parser by the transitions: expand - shift, expand - reduce, reduce - shift, reduce - reduce. The situation (ii) corresponds to stay - shift and stay - reduce and the situation (iii) corresponds to expand - stay and reduce - stay. Finally, one of the processors will perform the transitions of accepting and rejecting the input word. The nondeterministic behavior of the general left bidirectional parser is realized in the parallel algorithm by introducing those backtracking points in `action1(P1)` and `action2(P2)`. If no backtracking step can be performed, then the variable *exit* is set to true, so the input word is not accepted by the parser (and of course, this is not contained into the language of the input grammar).

So, (PAR_LEFT) is correct. ■

We may note that it is possible to store into the local memories of P1 and P2 the input word (and deleting it from the common memory). This does not change the original implementation issues because `w` is accessed in a read style. In this case the only variables which stored into the common memory are `i1` and `i2`.

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

We may describe another parallel algorithm which implements the **general right bidirectional parsing strategy.** Because it is similar to the general left bidirectional parsing strategy, we shall describe only the different parts.

The parallel algorithm (PAR_RIGHT) corresponding to the "right" model is ($G = (V_N, V_T, S, P)$ 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 hand 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= no(B → β), β is in Stack1 beginning with the 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
       backtracking step is needed;
       if (all the backtracking steps ended) and
         (still no reduce or expand action could be performed)
       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 top → α ∈ P, where r = no(top → α);
       if (there exists an r of this form) then begin
         push(Stack2,α);
         push(Output_tape2,r);
       end
     end;
     otherwise: begin
```

```
        backtracking step is needed;
        if (all the backtracking steps ended) and
          (still no reduce or expand action could be performed)
        then exit:=true
      end
end;
```

Similar results - as for PAR_LEFT - related to the termination and correctness of the above Algorithm (PAR_RIGHT) can be stated and proved.

4.3 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 problem, in $\mathcal{O}(n)$ time complexity (n being the length of the input word).

We start by giving an important result which establishes the relation between a context free grammar G and its reversed version $\widetilde{G}$.

Theorem 4.3.1 *Let $G = (V_N, V_T, S, P)$ be a context free grammar and let $\widetilde{G} = (V_N, V_T, S, \widetilde{P})$ be its reverse. Then, for any $A \in V_N$, and any derivation $\pi \in \{1, 2, ..., |P|\}^*$, we have:*

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

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

Proof We shall proceed by induction on $|\pi|$. The basis ($|\pi| = 1$) is obviously true.

We shall show only the direct implication of the equivalence.

Inductive Step: Let $A \underset{G,lm}{\overset{\pi}{\Longrightarrow}} w\,\gamma$ be the left most derivation π, ($|\pi| \geq 1$), where $w \in V_T^*$, $\gamma \in V_N \cdot V^* \cup \{\lambda\}$. $|\pi| \geq 1$ means that there exists a production numbered by r in G such that $\pi = \pi_1 r$. Now, we may rewrite the derivation as:

$$A \underset{G,lm}{\overset{\pi_1}{\Longrightarrow}} w_1\,B\,\gamma_2 \underset{G,lm}{\overset{r}{\Longrightarrow}} w\,\gamma, \text{ where } r = no(B \to w_2)\,\gamma_1,\ w_1\,w_2 = w,\ \gamma_1\,\gamma_2 = \gamma.$$

But $|\pi_1| < |\pi|$, so we can apply the inductive hypothesis to get:

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

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

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

The other implication may be similarly obtained. ■

4.3.1 $RR(k)$ grammars

The name $RR(k)$ is a shorthand for **R***ight to left scanning of the input constructing a* **R***ightmost derivation in reverse, using* **k** *symbols of lookahead* (see also Definition 2.1.14).

Definition 4.3.1 *Let G be a context free grammar and k be a natural number. We say that G is $RR(k)$ 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 $RR(0)$ grammar (respectively reduced grammar), then for any $A \in V_N$, there exists (respectively exactly) at most one production of the form $A \to \alpha \in P$. Certainly, if G is a reduced $RR(0)$ grammar, then its language is finite. That is why these grammars have no practical interest.

In [LeS68] it is pointed out that $RR(k)$ grammars "may be obtained from $LL(k)$ by reversing the roles of left and right", but no formal definition was given.

In order to define a parser for $RR(k)$ grammars, we shall give - for the beginning - some definitions and results which are "dually" obtained from similar ones for $LL(k)$ grammars.

Theorem 4.3.2 *Any $RR(k)$ grammar is unambiguous.*

Proof Using Definition 4.3.1 and a corresponding result from ([LeS68]). ■

Theorem 4.3.3 *If G is a right recursive grammar, then there exists no natural number k such as G be a $RR(k)$ grammar.*

Proof From Definitions 2.1.14 and 4.3.1. ■

Theorem 4.3.4 *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 From Definitions 2.1.14, 4.3.1 and the strict inclusion "$LL(k) \subset LL(k+1)$ ([[LeS68]]). ■

Lemma 4.3.1 *There exist context free languages which are not $RR(k)$ for any natural number k.*

Proof For example, $L = \{a^n\, c\, b^n,\ a^{2n}\, d\, b^n \mid n \geq 1\}$ cannot be $RR(k)$, for any natural number k. The proof can be done using a similar procedure as in Theorem 3.1.3. ■

In the following, we shall define a subclass of $RR(k)$ grammars (called $SRR(k)$) for which there exists an efficient algorithm for solving the problem 'G is a $SRR(k)$ grammar'. Furthermore, the subclass $SRR(1)$ coincides with $RR(1)$. We need also few auxiliary sets of words useful for defining the class of $SRR(k)$ grammars.

Definition 4.3.2 *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*

$$LAST_k(\alpha) = \{u \in V_T^* \mid |u| = k,\ \alpha \overset{*}{\underset{G}{\Longrightarrow}} v\,u,\ v \in V_T^*\} \cup \{u \in V_T^* \mid |u| < k, \alpha \overset{*}{\underset{G}{\Longrightarrow}} u\}$$

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

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

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

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

Definition 4.3.3 *A context free grammar $G = (V_N, V_T, S, P)$ is $SRR(k)$ if for any $A \in V_N$ and any two (different) productions $A \to \beta_1$, $A \to \beta_2$ the following statement holds:*

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

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 basic component.

Definition 4.3.4 *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 new character (a terminal symbol). The $PSRR_k(G)$* **parser** *is given by 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 $\underset{PSRR_k(G)}{\vdash\!\!\!\!-\!\!\!\!-}$) between configurations and it is given by:*

1. 1^0 *Expand:* $(\#u, \#\gamma\,A, \pi) \vdash (\#u, \#\gamma\,\beta, \pi\,r)$ *if* $r = no(A \to \beta)$ *and* $\#u^{(k)} \in LAST_k(\#PREVIOUS_k(A) \cdot \beta)$;
2. 2^0 *Reduce:* $(\#u\,a, \#\gamma\,a, \pi) \vdash (\#u, \#\gamma, \pi)$
3. 3^0 *Accept:* $(\#, \#, \pi) \vdash ACC$;
4. 4^0 *Rejection:* $(\#u, \#\gamma, \pi) \vdash REJ$ *if no transitions of type* 1^0, 2^0 *and* 3^0 *can be applied.*

Theorem 4.3.5 *Let G be a $SRR(k)$ grammar. Then the parser $PSRR_k(G)$ is deterministic, i.e. for any given configuration at most one transition can be applied.*

Proof The *expand* transitions are deterministic according to Definition 4.3.3, i.e. for an arbitrary configuration there exists at only one transition which can be applied. The rest of the transitions are deterministic, too. ■

Theorem 4.3.6 *For any $k \in \mathbf{N}^+$ the class of $SRR(k)$ grammars is strictly included in the class of $SRR(k+1)$ grammars.*

Proof The fact that every $SRR(k)$ grammar is also a $SRR(k+1)$ grammar is obvious. 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 (according to Definition 4.3.3). ■

Similar results as for $RR(k)$ grammars could be designed for $SRR(k)$ grammars. For instance, *any $SRR(k)$ grammar ($k \in \mathbf{N}$) is unambiguous*, and *a right recursive grammar is not a $SRR(k)$ grammar, for any $k \in \mathbf{N}^+$*. A similar result is:

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.

4.3.1.1 $RR(1)$ grammars

$RR(1)$ grammars proved to be useful in practice because - for them - the auxiliary sets $LAST_1$, $PREVIOUS_1$ (simply denoted, from now on, by $LAST$ and $PREVIOUS$) may be computed in polynomial time. According to Definition 4.3.2, we can rewrite $LAST$ and $PREVIOUS$ as follows ($\alpha \in V^+$ and $A \in V_N$):

$$LAST(\alpha) = \{a \mid a \in V_T,\ \alpha \stackrel{*}{\Longrightarrow} u\,a\} \cup \{\lambda \mid \alpha \stackrel{*}{\Longrightarrow} \lambda\};$$

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

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

Theorem 4.3.7 *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 statement is true:*

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

Proof According to Definition 4.3.1 and [LeS68] which proved a similar result for $LL(1)$ grammars. ■

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

Definition 4.3.5 *Let G be a context free grammar. Then:*

1. X `begin` A *iff* $\exists\ A \to \alpha\,X\,\beta \in P$ *and* $\alpha \stackrel{*}{\Longrightarrow} \lambda$;

2. X `end` A *iff* $\exists\ A \to \alpha X \beta \in P$ *and* $\beta \stackrel{*}{\Longrightarrow} \lambda$;

3. X `followed_by` Y *iff* $\exists\ A \to \alpha X \beta Y \gamma \in P$ *and* $\beta \stackrel{*}{\Longrightarrow} \lambda$;

4. X `terminal` Y *iff* $X,\ Y \in V_T$ *and* $X = Y$;

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

6. `last`=`terminal` $\circ$ `end`* $\circ$ `nonterminal`;

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

Theorem 4.3.8 *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 hold true:*

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

(ii) $a \in PREVIOUS(X)$ *iff* a `previous` X; $\lambda \in PREVIOUS(X)$ *iff* X `begin`* S.

Proof According to Definitions 4.3.2 and 4.3.5. ■

Example 4.3.1 *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.** $C \to C\,S\,;$

In order to check if this grammar is $LL(1)$, we compute the null nonterminal symbols, ($X \in V_N$ is a null symbol if $\exists\ X \stackrel{*}{\Longrightarrow} \lambda$). For our grammar the set of null symbols is $\{S, E, C\}$. We then compute the auxiliary binary relations:

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

So, the sets $LAST$ for the nonterminal symbols are:

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

We continue with the rest of our (last introduced) relations:

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

Hence, the sets $PREVIOUS$ corresponding to the nonterminal symbols are:

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

According to Theorem 4.3.7, 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.

It is the time to define the parser attached to a $RR(1)$ grammar.

Definition 4.3.6 *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 new character (a terminal symbol). The $PRR_1(G)$* **parser** *consists of 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 $\vdash_{PRR_1(G)}$) between configurations given by:*

1. 1^0 *Expand:* $(\#u,\#\gamma\,A,\pi)\vdash(\#u,\#\gamma\,\beta,\pi\,r)$ *if* $r=no(A\to\beta)$ *and* $\#u^{(1)}\in LAST(\#PREVIOUS(A)\cdot\beta)$;
2. 2^0 *Reduce:* $(\#u\,a,\#\gamma\,a,\pi)\vdash(\#u,\#\gamma,\pi)$;
3. 3^0 *Accept:* $(\#,\#,\pi)\vdash ACC$;
4. 4^0 *Rejection:* $(\#u,\#\gamma,\pi)\vdash REJ$ *if no transitions of type* 1^0, 2^0 *and* 3^0 *can be applied.*

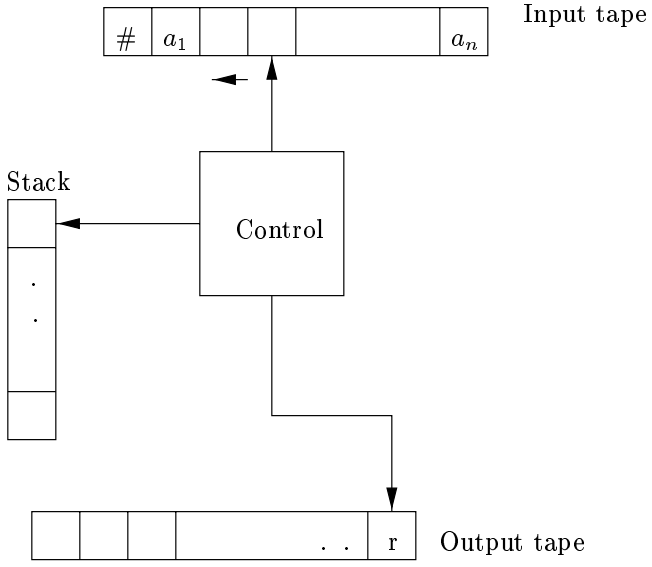


Figure 4.4. The $PRR_1(G)$ parser

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

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

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

Input: Any $RR(1)$ grammar $G = (V_N, V_T, S, P)$, the sets $LAST, PREVIOUS$ corresponding to the nonterminal symbols and a word $w \in V_T^*$;
Output: The right hand syntactical analysis if $w \in L(G)$; otherwise the message: 'w is not accepted'.
Method: Assume that we have at our disposal the following procedures:

- `pop(stack,top)` - as a result, the value of `top` will coincide with the value of the first symbol from `stack`;
- `push(stack,X)` - pushes in the top of `stack` the value of `X`;
- `push(Output_tape,r)` - pushes in the top of `Output_tape` the value of `r`.

Then the main program is:

```
begin
   read(w); push(stack,"#S"); i:= |w|+1;
   accept:=false; is_over:=false;
   put into 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= no(top→ α) 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 right hand syntactic analysis ',Output_tape)
    else write('w is not accepted.')
end.
```

Theorem 4.3.9 *(correctness and complexity of Algorithm (Parsing-$RR(1)$))*
Algorithm (Parsing-$RR(1)$) is correct and has the time complexity $\mathcal{O}(|w|)$, where w is the input word.

Proof The correctness follows directly from Definition 4.3.1 and [LeS68]. The number of iterations of the loop "`repeat`... `until`" is $\mathcal{O}(m \cdot |w|)$, where m is a constant depending on 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. Any reduce action implies the reading of one letter from w. Therefore the time complexity of Algorithm (Parsing-$RL(0)$) is $\mathcal{O}(|w|)$. ■

Example 4.3.2 *We reconsider the grammar of Example 4.3.1. 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 hand 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:

b - `begin`*, e -* `end`*, a -* `statement` *and* ; *-* `end of statement`*.*

4.3.2 $RL(k)$ grammars

The name $RL(k)$ is a shorthand for **R***ight to left scanning of the input constructing a* **L***eftmost derivation in reverse, using* **k** *symbols of lookahead* (Definition 2.1.15).

Definition 4.3.7 *Let G be a context free grammar and k be a natural number. We say that G is an $RL(k)$ grammar if $\widetilde{G}$ is an $LR(k)$ grammar. A language L is $RL(k)$ if there exists an $RL(k)$ grammar generating L.*

An example of an $RL(0)$ grammar can be found in [Knu65], but the only "definition" for $RL(0)$ grammars was that these are obtained from $LR(0)$ by an "asymmetric property". Later ([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 also need first some definitions and results which are "dual" to similar ones for $LR(k)$ grammars.

Theorem 4.3.10 *Any $RL(k)$ grammar is unambiguous.*

Proof Using Definition 4.3.7 and a similar result for $LR(k)$ grammars ([Knu65]). ■

Theorem 4.3.11 *([Knu65]) There exist $RL(0)$ languages which cannot be $LR(k)$, $\forall\ k \in \mathbf{N}$.*

Proof Let G be the grammar:

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

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

4.3.2.1 $RL(0)$ grammars

Definition 4.3.8 *Let $G = (V_N, V_T, S, P)$ be a context free grammar and $\blacksquare \notin V$ a new symbol (called* **point***). An $RL(0)$* **item** *for G is a construction $A \to \beta_1 \blacksquare \beta_2$, where $A \to \beta_1\beta_2 \in P$. The set of all items of G is denoted by $I(G)$.*

Example 4.3.3 *For the production $S \to x_1\,x_2 \ldots x_n$ there exist $n+1$ items:*

$$S \to x_1\,x_2 \ldots x_n \blacksquare,\ S \to x_1\,x_2 \ldots x_{n-1} \blacksquare x_n,\ \ldots\ S \to \blacksquare x_1\,x_2 \ldots x_n$$

Definition 4.3.9 *An $RL(0)$ item of the form $A \to \blacksquare\beta$ is called* **complete** *(the point is at the beginning of the production).*

Definition 4.3.10 *A* **viable suffix** *for G is a word $\gamma \in V^*$ for which there exists a derivation $S \underset{lm}{\overset{*}{\Longrightarrow}} u\,A\,\alpha \underset{lm}{\Longrightarrow} u\,\beta\alpha$, and γ is a suffix for $\beta\,\alpha$ (i.e. there exists $\gamma' \in V^*$ such that $\beta\,\alpha = \gamma'\,\gamma$).*

Example 4.3.4 *Let us consider the left hand derivation:*

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

According to Definition 4.3.10, the viable suffixes for $a\,c\,d\,B$ are λ, B, $d\,B$, $c\,d\,B$ and $a\,c\,d\,B$.

Definition 4.3.11 *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 4.3.5 *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 4.3.12 *(characterization of $RL(0)$ grammars)*
A reduced grammar $G = (V_N, V_T, S, P)$ (and in which S does not occur in the right hand 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 4.3.13 *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 think that some hints of that proof may be useful for what follows. Starting from the reduced context free grammar $G = (V_N, V_T, S, P)$, a nondeterministic finite automaton with $\lambda-$transitions can be derived which accepts the set of all viable suffixes. Let $M = (I, V, \delta, q_0, I)$ be such an automaton where:

- $I = \{q_0\} \cup \{A \to \beta_1 \centerdot \beta_2 \mid A \to \beta_1\beta_2 \in P\}$
- 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) $\delta(A \to \alpha\, a \centerdot \beta, \lambda) = \delta(A \to \alpha\, X \centerdot \beta, Y) = \emptyset,\ Y \in V_N,\ X \neq Y,\ a \in V_T$.

q_0 is the initial state, that is 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 hand 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 4.3.6 *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 4.3.12. The two conditions from Theorem 4.3.12 may be "translated" into the automaton graph as:

(i) the automaton graph does not contain two vertices $A \to \alpha$, $B \to \beta$ ($A \to \alpha \neq B \to \beta$), such as the paths from q_0 to these vertices be labeled with the same word;

(ii) the automaton graph does not contain two vertices $A \to \alpha$, $B \to \beta_1\, a\, \beta_2$, $a \in V_T$ such as the paths from q_0 to these vertices be labeled with the same word.

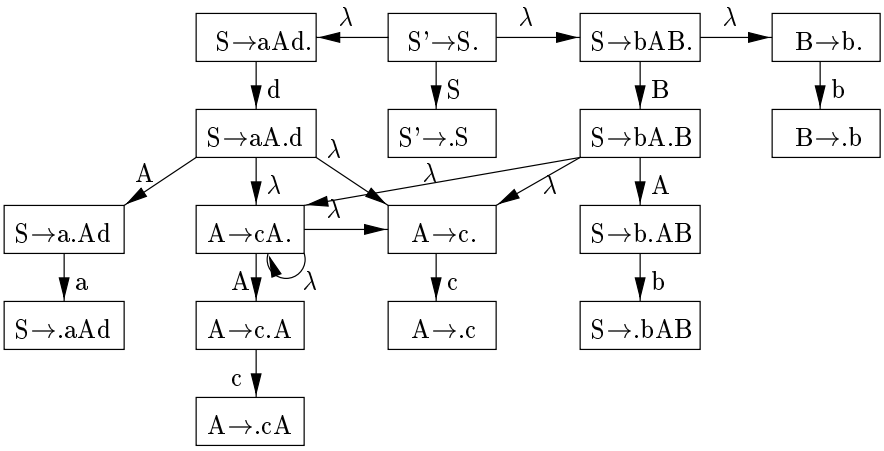


Figure 4.5. The automaton with λ-transitions of viable suffixes

Let us note that the graph for the automaton M satisfies the previous conditions (i) and (ii). Hence G is a $RL(0)$ grammar.

However, the conditions considered in Example 4.3.6 are hard to check on this automaton with $\lambda-$transitions. For each M, we may attach an equivalent (i.e. $L(M) = L(M')$) deterministic automaton $M' = (T, V, \delta', t_0, T)$, where $T \subseteq 2^I$ (one state from M' corresponds to a subset of items from M), 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.

We know that 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 $\blacksquare$.

These two conditions may be easily checked for M'. In the following, we shall indicate the way of constructing M' (called the $RL(0)$ **automaton** of a grammar G) starting from G. The function *closure* has as input a set of items t and returns as output all the states (sets of items) accessible from t through $\lambda-$paths.

The Algorithm (Closure)

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

```
function closure(t):T;
(* T is a notation for a subset of I *);
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 4.3.2 *Let $M = (I, V, \delta, q_0, I)$ be the automaton containing $\lambda-$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 give an algorithm for constructing the $RL(0)$ automaton for grammar G.

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

Input: $G = (V_N, V_T, S, P)$, any 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
  t_0:=closure(S' → S.); T:= {t_0}; marcat[t_0]:=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 4.3.3 *The Algorithm (RL(0)-Automaton) is correct, i.e. M' is a deterministic automaton equivalent to M (i.e. $L(M') = L(M)$).*

Example 4.3.7 *Consider the same grammar as in Example 4.3.6. Using Algorithm (RL(0)-Automaton) we shall construct the deterministic automaton M' (the numbering of states in M' will be done in a breadth first search manner):*

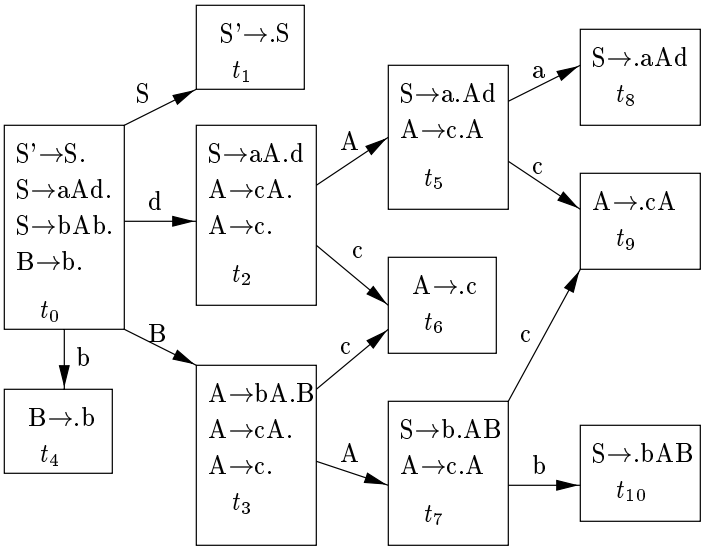


Figure 4.6. The deterministic automaton M'

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

Example 4.3.8 *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 given by:

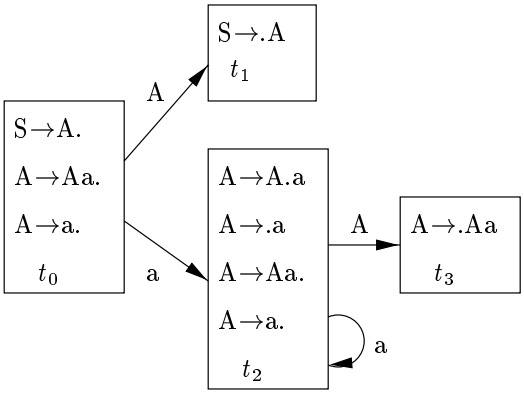


Figure 4.7. The deterministic automaton M'

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

We can adapt the $LR(0)$ language characterization theorem to characterize $RL(0)$ languages.

Theorem 4.3.14 *($RL(0)$ 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 $\widetilde{L} = L_0 L_1^*$.

Proof From Theorem 13.3.1, [Har78]. ■

The parser attached to a $RL(0)$ grammar may now be introduced.

Definition 4.3.12 *Let $G = (V_N, V_T, S, P)$ be a $RL(0)$ grammar and $M' = (T, V, \delta', t_0, T)$ be 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 new character (a terminal symbol). The $RL(0)$* **parser** *(denoted by $PRL_0(G)$) is given by $(\mathcal{C}_0, \vdash)$, 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 $\vdash_{PRL_0(G)}$) between configurations and it is given by:*

1. 1^0 *Shift:* $(\#u\,a, t\,\sigma, \pi) \vdash (\#u, t'\,a\,t\,\sigma, \pi)$ *if* $\delta'(t, a) = t'$;
2. 2^0 *Reduce:* $(\#u, t'\,\sigma\,t\,\sigma, \pi) \vdash (\#u, t''\,A\,t\,\sigma, r\,\pi)$ *if* $A \to \cdot\beta \in t'$, $|\sigma'\ t'| = |\beta|$, $r = no(A \to \beta)$, $t'' = \delta'(t, A)$;
3. 3^0 *Accept:* $(\#, \sigma, \pi) \vdash ACC$ *if* $S' \to \cdot A \in t_1$;
4. 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. The main 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'. The output tape will contain the left hand syntactic analysis. It returns "ACC" or "REJ" depending on whether the input word is accepted or not.

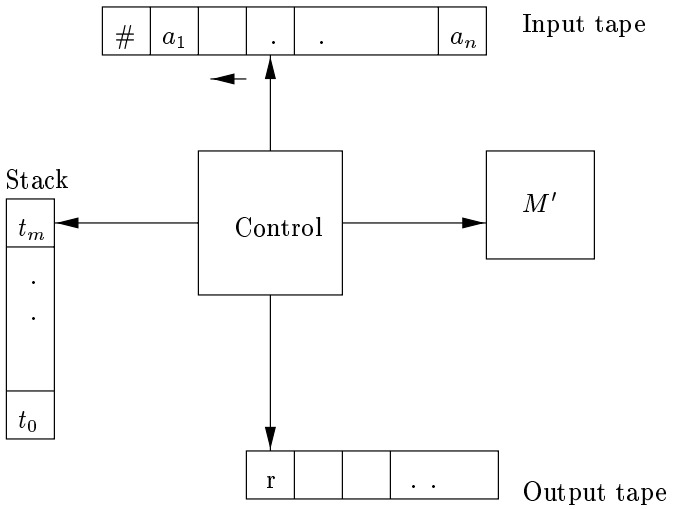


Figure 4.8. The $RL(0)$ parser

The parsing algorithm for $RL(0)$ grammars.

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

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

- `pop(stack,top)` - the computed value of `top` equals the value of the first symbol of `stack` (without removing the top of the `stack`);
- `push(stack,X)` - push into the top of the `stack` the value of `X`;
- `push(Output_tape,r)` - push into the top of `Output_tape` the value of `r`.

The main program is:

```
begin
   read(w);push(stack,t0);i:=|w|+1;
   accept:=false;is_over:=false;
   the input tape contains 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 → ▪X1 X2 ... Xm ∈t) then begin
         if (A = S') then begin
           /* accept action */
           push(Output_tape,no(S' → X1 X2 ... Xn));
           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,no(A → X1 X2 ... Xn))
         end
```

```
      end
      else is_over:=true /* reject action */
   until (is_over=true);
   if (accept=true) then
     write('w is accepted and has left hand syntactic analysis ', Output_tape)
   else write('w is not accepted.')
end.
```

Theorem 4.3.15 *(correctness and complexity of Algorithm (Parsing-$RL(0)$))*
Algorithm (Parsing-$RL(0)$) is correct and has the time complexity $\mathcal{O}(|w|)$, where w is the input word.

Proof The correctness follows directly from Definition 4.3.7 and [Knu65]. The number of iterations of the loop "`repeat... until`" is $\mathcal{O}(m \cdot |w|)$, where m is a constant depending of G. During an iteration, Algorithm (Parsing-$RL(0)$) performs a *shift* or a *reduce* action. The number of consecutively reduce actions is finite because the input grammar is finite. A shift action implies the reading of one letter from w. So the time complexity of Algorithm (Parsing-$RL(0)$) is $\mathcal{O}(|w|)$. ■

Example 4.3.9 *Reconsider the grammar in Examples 4.3.6 and 4.3.7. Let $w = b\,c\,c\,b$ be the input word for Algorithm (Parsing-$RL(0)$). We have:*
$(\#b\,c\,c\,b, t_0, \lambda) \overset{1^0}{\vdash} (\#b\,c\,c, t_4\,b\,t_0, \lambda) \overset{2^0}{\vdash} (\#b\,c\,c, t_3\,B\,t_0, [5]) \overset{1^0}{\vdash} (\#b\,c, t_6\,c\,t_3\,B\,t_0, [5]) \overset{2^0}{\vdash}$
$(\#b\,c, t_7\,A\,t_3\,B\,t_0, [4,5]) \overset{1^0}{\vdash} (\#b, t_9\,c\,t_7\,A\,t_3\,B\,t_0, [4,5]) \overset{2^0}{\vdash} (\#b, t_7\,A\,t_3\,B\,t_0, [3,4,5])$
$\overset{1^0}{\vdash} (\#, t_{10}\,b\,t_7\,A\,t_3\,B\,t_0, [3,4,5]) \overset{2^0}{\vdash} (\#, t_1\,S\,t_0, [2,3,4,5]) \overset{3^0}{\vdash} ACC$
Therefore, $w \in L(G)$ and the left hand syntactic analysis for it is $\pi = [2,3,4,5]$.

4.3.2.2 $SRL(1)$ grammars

There exist practical cases where, for a given context free grammar G which is not an $RL(0)$ grammar, the *conflicts* reduce - reduce, reduce - shift (of the corresponding $RL(0)$ automaton) can be solved. *Simple* $RL(1)$ grammars (denoted by $SLR(1)$) are needed.

Definition 4.3.13 *The context free grammar $G = (V_N, V_T, S, P)$ is $SRL(1)$ if for any state $t \in T$ of the $RL(0)$ automaton $M' = (T, V, \delta, t_0, T)$ defined as the output of Algorithm ($RL(0)$-Automaton), the following statements hold true:*

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

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

Next, we shall define a relation $ACTION : T \to V_T \cup \{\#\}$ (sometimes called an $SRL(1)$-table), which will be used to remove the conflicts reduce - reduce, reduce - shift. This relation is constructed using the $RL(0)$ (corresponding) automaton and the sets $PREVIOUS$:

The Algorithm $SRL(1)$-table:

Input: Any context free grammar $G = (V_N, V_T, S, P)$, the $RL(0)$ corresponding 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 = no(A → α);
      if (λ ∈ PREVIOUS(A)) then
        ACTION(t,#):=reduce_r, where r = no(A → α)
    end;
    for (all B → β a▪γ ∈ t, a ∈ V_T) do
      ACTION(t,a):=shift_k, where t_k = δ(t,a);
    if (S' → ▪S ∈ t) then
      ACTION(t,#):=ACC_r, where r = no(S' → S)
  end
end.
```

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 of Definition 4.3.13 hold and then G is a $SRL(1)$ grammar (and vice-versa).

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

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

```
begin
  read(w);push(stack,t_0);i:=|w|+1;
  accept:=false;is_over:=false;
  the input tape contains 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_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 r-th production;
            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 left hand syntactic analysis ', Output_tape)
    else write('w is not accepted.')
end.
```

Theorem 4.3.16 *(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 4.3.13, the construction of $ACTION$ and following the same procedure as in the proof of Theorem 4.3.15. ■

Example 4.3.10 *Let us consider the context free grammar G which generates all the arithmetic expressions (over $+$, $*$):*

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

It can be checked that G is not an $RL(0)$ grammar, but it is an $SRL(1)$ grammar ([AnG95]).

4.3.2.3 $RL(1)$ grammars

Definition 4.3.14 *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 4.3.15 *A* **viable suffix** *for G is a word $\gamma \in V^*$ for which there exists a derivation $S \underset{lm}{\overset{*}{\Longrightarrow}} u\,A\,\alpha \underset{lm}{\Longrightarrow} u\,\beta_1\,\beta_2\,\alpha$, γ being 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 4.3.17 *(characterization of $RL(1)$ grammars)*
A reduced context free 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 of the form:

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

Proof According to Definition 4.3.7, the proof is similar to the corresponding result for $LR(1)$ grammars ([Knu65]). ■

For describing the so called $RL(1)$ automaton, the function called `closure(I)` needs to be described. 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 a given grammar G.

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

Input: $G = (V_N, V_T, S', P \cup S' \to S)$, a context free grammar;
Output: $M = (T, V, \delta, t_0, T)$, the $RL(1)$ deterministic automaton associated to G;
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 4.3.4 *The automaton M (the output of Algorithm ($RL(1)$-Automaton)) is deterministic and accepts 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.*

Proof According to Definition 4.3.7, the proof is similar to the corresponding result for $LR(1)$ grammars ([Knu65]). ■

Example 4.3.11 *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 ([AnG95]).

The way of constructing the $RL(1)$-table is the only possible difference between $RL(1)$ and $SRL(1)$.

The Algorithm $RL(1)$-table:

Input: Any context free grammar $G = (V_N, V_T, S, P)$, the $RL(1)$ equivalent 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 = no(A → α);
    if ((#, A → •α) ∈ t and A ≠ S') then
      ACTION(t, #):=reduce_r, where r = no(A → α);
    if ((#, S' → •S) ∈ t) then
      ACTION(t, #):=ACC_r, where r = no(S' → S);
    for (all a ∈ V_T ∪ {#}) do
      if (ACTION(t, a) = ∅) then ACTION(t, a) = REJ
  end
end.
```

In the next subsection we shall see how is it possible that the number of states of the corresponding automaton of the input grammar will be the same as the number of states of the $RL(0)$-automaton. This subclass is called $LARL(1)$ grammars (Look Ahead $RL(1)$).

4.3.2.4 $LARL(1)$ grammars

For an $RL(1)$ automaton it may happen that some states have the same "values" on the first component of $RL(1)$ items. These states are somehow "equivalent".

Definition 4.3.16 *Let t be a state in an $RL(1)$ automaton corresponding to a 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.*
$Ker(t) = \{A \to \alpha_1 \centerdot \alpha_2 \mid \exists\ (a, A \to \alpha_1 \centerdot \alpha_2) \in t\}$.

Example 4.3.12 $Ker(\{(a, A \to \alpha_1 \centerdot \alpha_2),\ (b, A \to \alpha_1 \centerdot \alpha_2),\ (c, B \to \beta_1 \centerdot \beta_2)\})$ *equals* $\{A \to \alpha_1 \centerdot \alpha_2,\ B \to \beta_1 \centerdot \beta_2\}$.

Definition 4.3.17 *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 an $RL(1)$ automaton is a set of $RL(1)$ items, we may define the "union" of two states.

Definition 4.3.18 *Let $t_1 = \{(M_1, K_1)\}$ and $t_2 = \{M_2, K_2)\}$ be two equivalent states of an $RL(1)$ automaton, i.e. $K_1 = K_2$. Then we denote $t_1 \cup t_2 = \{(M_1 \cup M_2, K_1)\}$.*

Example 4.3.13 *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 4.3.19 *Let $G = (V_N, V_T, S, P)$ be an $RL(1)$ grammar and let $M = (T, V, \delta, t_0, T)$ be the corresponding $RL(1)$ automaton. We say that G is an $LARL(1)$ 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 *in the sense of Definition 4.3.13).*

The algorithms related to $LARL(1)$ grammars are similar to the corresponding algorithms concerning $RL(1)$ grammars. The main difference concerns the algorithm for constructing the $LARL(1)$ automaton for the input grammar G.

First, we compute the $RL(1)$ automaton $M = (T, V, \delta, t_0, T)$ for the $RL(1)$ grammar $G = (V_N, V_T, S, P)$. Let us denote $T = \{t_0, t_1, ..., t_n\}$. Then we determine the equivalent states (in the sense of Definition 4.3.17). According to Definition 4.3.18, for the equivalent states, the union operation is made. Thus, 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 (in the sense of Definition 4.3.13), then we say that G is not a $LARL(1)$ grammar (Definition 4.3.18). 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'.

4.3.3 SIP grammars

In this section, we present formal definition for SIP grammars, which can be viewed as "mirroring" the precedence grammars (see Definitions 2.1.16 and 2.1.17).

Definition 4.3.20 *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 4.3.21 *For any $G = (V_N, V_T, S, P)$ context free grammar without null productions the following relations (called* **inverse precedence relations***) $<\!\!\cdot \subset V_T \times V$ and $\doteq$, $\cdot\!>\!> \subseteq V \times V$ are:*

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

Let # be a new terminal symbol. We extend the previous binary relations to that symbol such as:
$\#<\!\!\cdot X$ iff $\exists\ S \overset{+}{\Longrightarrow} X\,\alpha$ and $X \cdot\!>\!>\#$ iff $\exists\ S \overset{+}{\Longrightarrow} \alpha\, X$.
According to Definitions 2.1.16, 4.3.20 and 4.3.21, the following statements immediately hold:

- $X <\!\!\cdot Y$ in G iff $Y \cdot\!>\!> X$ in $\widetilde{G}$;
- $X \doteq Y$ in G iff $Y \doteq X$ in $\widetilde{G}$ (*egal* coincides to $\doteq$);
- $X \cdot\!> Y$ in G iff $Y <\!\!\cdot X$ in $\widetilde{G}$.

Therefore, a context free grammar G without null productions, in which the binary relations $<\!\!\cdot$, $\doteq$, $\cdot\!>\!>$ are disjoint, may be called an **inverse precedence grammar.**

Theorem 4.3.18 *Let $G = (V_N, V_T, S, P)$ be a reduced context free grammar without null productions and let*

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

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

1. $u_k <<\cdot Y_1$;
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 4.3.22 *Let $G = (V_N, V_T, S, P)$ be an inverse precedence grammar and let $<<\cdot$, $\doteq$, and $\cdot>>$ be 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 new character (a terminal symbol). The* **simple inverse precedence parser** *(denoted by $SIPP(G)$) is defined by 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 $\vdash_{SIPP(G)}$) between configurations, it is 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* $\beta = \beta_1 ... \beta_m$, $u^{(1)} <<\cdot \beta_1$, $\beta_k \doteq \beta_{k+1},\ \forall\ k = \overline{1, m-1}$, $\beta_m \cdot >>^{(1)} \gamma$, $r = no(A \to \beta)$;

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 $SIPP(G)$ resembles the parser $PRL_0(G)$. It only has auxiliary parsing informations (the relations $<<\cdot$, $\doteq$, $\cdot>>$).

Example 4.3.14 *Let G be a context free grammar given by:*

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

We have

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

Therefore ($<<\cdot$, $\doteq$, $\cdot>>$ are disjoint) 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 derive the transition sequence:

$$(\#a\,a\,c\,c\,b\,c\,b, \#, \lambda) \overset{1^0}{\vdash} (\#a\,a\,c\,c\,b\,c, b\#, \lambda) \overset{1^0}{\vdash} (\#a\,a\,c\,c\,b, c\,b\#, \lambda) \overset{2^0}{\vdash}$$

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

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

The parsing algorithm for simple inverse precedence grammar below uses the predefined procedures `pop`, `push` already known.

The Algorithm (Parsing-SIP)

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

```
begin
   read(w);push(stack,#);i:=|w|+1;
   accept:=false;is_over:=false;
   put into 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 in stack for which:
           Yk≐Yk+1, ∀ i = 1,m-1, Ym≐X1;
           find a production of the form r= no(A → Y1 Y2 ... Ym);
           if (there 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 Y_1 Y_2 ... Y_m 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 left hand syntactic analysis ', Output_tape)
  else write('w is not accepted.')
end.
```

Theorem 4.3.19 *(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 4.3.15. ■

4.4 Deterministic bidirectional parsing for context free languages

In this section we shall present several ways in which some subclasses of context free grammars may be combined such as to obtain deterministic (and linear) parallel algorithms for solving the corresponding membership problem.

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

Definition 4.4.1 *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 an* $LL(k)$ *and an* $RL(0)$ *grammar;*
2. G *is a* $LL(k) - SRL(1)$ *grammar if* G *is an* $LL(k)$ *and an* $SRL(1)$ *grammar;*
3. G *is a* $LL(k) - RL(1)$ *grammar if* G *is an* $LL(k)$ *and an* $RL(1)$ *grammar;*
4. G *is a* $LL(k) - LARL(1)$ *grammar if* G *is an* $LL(k)$ *and an* $LARL(1)$ *grammar;*
5. G *is a* $LL(k) - SIP$ *grammar if* G *is an* $LL(k)$ *and an* SIP *grammar;*

6. *G is a $LR(0)-RR(k)$ grammar if G is an $LR(0)$ and an $RR(k)$ grammar;*
7. *G is a $SLR(1)-RR(k)$ grammar if G is an $SLR(1)$ and an $RR(k)$ grammar;*
8. *G is a $LR(1)-RR(k)$ grammar if G is an $LR(1)$ and an $RR(k)$ grammar;*
9. *G is a $LALR(1)-RR(k)$ grammar if G is an $LALR(1)$ and an $RR(k)$ grammar;*
10. *G is a $SP-RR(k)$ grammar if G is an SP and an $RR(k)$ grammar.*

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 4.4.1 *The following statements hold true:*

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

Proof Directly from Definitions 4.3.1, 4.3.7, 4.3.20 and 4.4.1. ■

It is obvious that all the languages associated to the grammars in Definition 4.4.1 are deterministic context free languages. Using a "mirroring" strategy, we can easily extend some known results ([AhU72], [Har78], [HoU79], [Sal73]) concerning e.g. inclusions and hierarchies for classical subclasses of deterministic context free languages. Therefore, the following relations hold ($\forall\ k \in \mathbf{N}$):

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

Note that the first five classes of grammars (Definition 4.4.1) "use" a left hand bidirectional strategy and the last five "use" a right hand bidirectional strategy.

This deterministic parallel approach is similar to the general parallel approach, the only difference being the absence of the backtracking steps. Thus the "kernel" of the parallel iteration corresponding to the left hand bidirectional strategy is (we use the same considerations as for 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 the corresponding sequential algorithms for syntactic analysis (i.e. associated to the classical subclasses of grammars and to the subclasses described in Section 4.3, such as Algorithms Parsing-RR(1), Parsing-RL(0), Parsing-SLR(1), Parsing-SIP). This time, instead of exponential sequential running time, we have a linear running time for the procedures `action1` and `action2`, because backtracking steps are not necessary. The linear time complexity follows directly from Theorems 4.3.9, 4.3.15, 4.3.16, 4.3.19 and from known 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 analyzers for the specific subclasses of context free grammars.

Theorem 4.4.1 *(the complexity of the deterministic parallel algorithms)*
Let us denote by $T_1(n)$, $T_2(n)$ the running time of the sequential syntactic analyzers from Section 4.3, 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 have supposed 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 $t(n)$ is linear follows from the linear complexity of the deterministic parsers associated to the considered subclasses of grammars. ■

Example 4.4.1 *Let $G = (\{S', S, B, C\}, \{a, b, e, ;\}, S', P)$ be a context free grammar, 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 prove that G is a $LL(1) - LARL(1)$ grammar. Compute first 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 those given by 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.

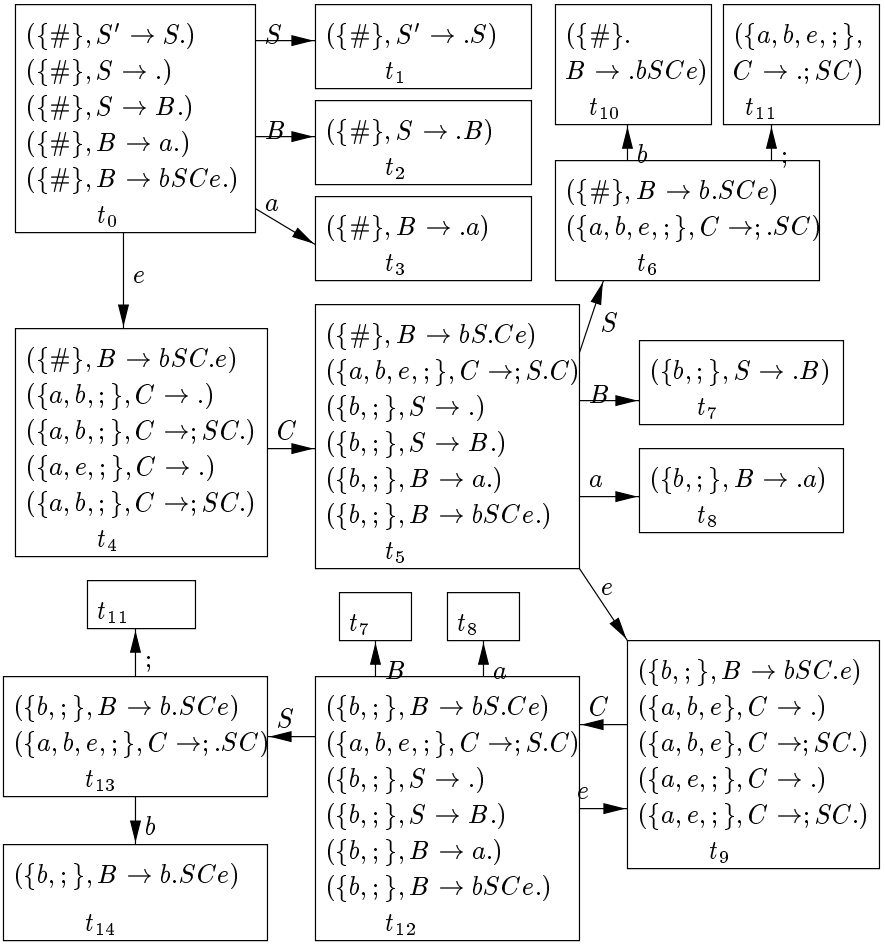


Figure 4.9. $RL(1)$ automaton for G

In Figure 4.9, the arcs (t_{12}, t_7), (t_{12}, t_8) *and* (t_{13}, t_{11}) *are depicted differently only because of the picture size.*

It can be checked looking at the $RL(1)$ *automaton that* G *is a* $RL(1)$ *grammar (Theorem 4.3.17). Furthermore, the following pairs of states are equivalent (Definition 4.3.17):*

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

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 represents a 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 hand 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 vice-versa. 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 until 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:*

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

We can extend h_1 to words of arbitrary length using the function $h : (V \cup T)^* \to V^*$, given by:

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

We remind the reader that T is the set of states of 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 hand syntactic analysis $[1, 3, 5, 3, 4, 7, 3, 5, 3, 4, 6, 6]$.

4.5 Conclusions

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

The main complexity result is Theorem 4.4.1 (see also Example 4.1.2).

Open problems:

- find new subclasses of deterministic parallel algorithms for bidirectional parsing;
- estimate more precisely the running time of the deterministic parallel algorithm presented in Section 4.4;
- find further closure properties of the subclasses for described languages.

Chapter 5

Up-to-up bidirectional parsing for context free grammars

In this chapter we describe some subclasses of context free grammar for which there exists a parallel approach useful for solving the membership problem ([AGK99]). We combine the classical style of LR parsers attached to a grammar G with a "mirror" process for G by analyzing the input word from both sides and using two processors.

In the first section we present the *general up-to-up bidirectional parser* (it can analyze any context free language) which has a nondeterministic nature.

A general SIMD model for describing the up-to-up bidirectional parsing is presented in the next section.

Our general up-to-up bidirectional parser can be also used as a deterministic model for the known $LR(k)$ and $RL(k)$ parsers (the third section). The membership problem may be solved in linear time complexity with a parallel algorithm.

5.1 The general up-to-up bidirectional parser

In this section we define a new parser for the class of context free languages. The input word is analyzed from both sides (as in [AnK99a]), but the parse strategy is view in an "up-to-up" manner (LR and RL styles are combined for this parallel strategy). The associated derivation tree corresponding to the input word w is down to up traversed by both processors, that is from the leaves to its root in our view. Only one processor will be "strongly" active after the parallel algorithm "meets" the "middle" of the input word:

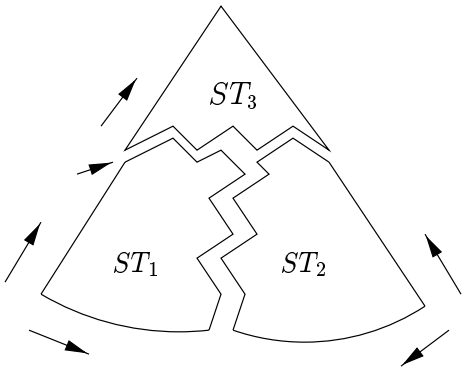


Figure 5.1. Strategy for up-to-up bidirectional parsing

The "derivation forests" ST_1 and ST_2 will be parsed in parallel and finally, the subtree ST_3 - in a sequential way.

Our bidirectional parser will be based on the following definitions and notations.

Notation 5.1.1 *Let $G = (V_N, V_T, S, P)$ be a context free grammar and V' be the set $V_N \times \mathbf{N} \times \mathbf{N}$. Let $h : V \cup V' \to V^*$ be the function given by:*

1. $h(X) = X, \ \forall \ X \in V$;
2. $h(X_{b,e}) = X, \ \forall \ X_{b,e} \in V'$;

This function can be easily extended to an homomorphism $h : (V \cup V')^* \to V^*$ such as:

1. $h(\lambda) = \lambda$;
2. $h(X_1 X_2 ... X_n) = h(X_1) \cdot h(X_2) \cdot ... \cdot h(X_n), \ \forall \ X_1, \ X_2, \ ..., \ X_n \in V \cup V',$

$\forall \ n \geq 2.$

The notation of $A_{b,e}$ signifies that A is the label of the derivation subtree of root v in the forests ST_1 or ST_2 (Figure 5.1), and also that the frontier has the corresponding right most derivation $[r_b, r_{b+1}, ..., r_e]$ (further details in Lemma 5.1.4 and Example 5.1.1).

Definition 5.1.1 *Let $G = (V_N, V_T, S, P)$ be a context free grammar. Let $\mathcal{C} \subseteq \{s_1, s_2\} \times \{1, 2, ..., |P|\}^* \times \#(V \cup V')^* \times \#V_T^* \# \times (V \cup V')^* \# \times \{1, 2, ..., |P|\}^* \times \{1, 2, ..., |P|\}^* \cup \{ACC, REJ\}$ be the set of all possible configurations, where $\#$ is a new character (a terminal symbol). The* **general up-to-up bidirectional parser** *(denoted by $G_u BP(G)$) is given by the pair $(\mathcal{C}_0, \vdash)$, where $\mathcal{C}_0 = \{(s_1, \lambda, \#, \#w\#, \#, \lambda, \lambda) \mid w \in V_T^*\} \subseteq \mathcal{C}$ is called the set of* **initial configurations**. *The first component is the state, the following, excepting the last one component of a configuration are used for storing the partial syntactic analysis. The last component is used for storing the final 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 $\underset{G_u BP(G)}{\longmapsto}$) between configurations is given by:*

1^0 *Shift-Shift:*

$(s_1, \pi_1, \#\alpha, \#a\,u\,b\#, \beta\#, \pi_2, \lambda) \vdash (s_1, \pi_1, \#\alpha\,a, \#u\#, b\,\beta\#, \pi_2, \lambda)$;

2^0 *Reduce-Shift:*

$(s_1, \pi_1, \#\alpha\,\beta, \#u\,b\#, \gamma\#, \pi_2, \lambda) \vdash (s_1, r_1\pi_1, \#\alpha\,A_{b',e'}, \#u\#, b\,\gamma\#, \pi_2, \lambda)$,
$r_1 = no(A \to h(\beta))$, $e' = |\pi_1|+1$, $b' = \min\{|\pi_1|+1, \min\{b'' \mid C_{b'',e''} \in \beta\}\}$.
If $h(\beta)$ *contains no nonterminal symbols, then* $b' = |\pi_1| + 1$;

3^0 *Shift-Reduce:*

$(s_1, \pi_1, \#\alpha, \#a\,u\#, \gamma\,\beta\#, \pi_2, \lambda) \vdash (s_1, \pi_1, \#\alpha\,a, \#u\#, B_{b,e}\,\beta\#, r_2\pi_2, \lambda)$,
$r_2 = no(B \to h(\gamma))$, $e = |\pi_2| + 1$, $b = \min\{|\pi_2| + 1, \min\{b' \mid D_{b',e'} \in \gamma\}\}$.
If $h(\gamma)$ *contains no nonterminal symbols, then* $b = |\pi_2| + 1$;

4^0 *Reduce-Reduce:*

$(s_1, \pi_1, \#\alpha\beta, \#u\#, \varepsilon\gamma\#, \pi_2, \lambda) \vdash (s_1, r_1\pi_1, \#\alpha A_{b_1,e_1}, \#u\#, B_{b_2,e_2}\gamma\#, r_2\pi_2, \lambda)$,
where $r_1 = no(A \to h(\beta))$, $r_2 = no(B \to h(\varepsilon))$, $e_1 = |\pi_1|+1$, $e_2 = |\pi_2|+1$,
$b_1 = \min\{|\pi_1| + 1, \min\{b_1' \mid C_{b_1',e_1'} \in \beta\}\}$ *and*
$b_2 = \min\{|\pi_2| + 1, \min\{b_2' \mid D_{b_2',e_2'} \in \varepsilon\}\}$;

5^0 *Shift-Stay:*

$(s_1, \pi_1, \#\alpha, \#a\,u\#, \beta\#, \pi_2, \lambda) \vdash (s_1, \pi_1, \#\alpha\,a, \#u\#, \beta\#, \pi_2, \lambda)$;

6^0 *Reduce-Stay:*

$(s_1, \pi_1, \#\alpha\,\beta, \#u\#, \gamma\#, \pi_2, \lambda) \vdash (s_1, r_1\pi_1, \#\alpha\,A_{b',e'}, \#u\#, \gamma\#, \pi_2, \lambda)$,
$r_1 = no(A \to h(\beta))$, $e' = |\pi_1|+1$, $b' = \min\{|\pi_1|+1, \min\{b'' \mid C_{b'',e''} \in \beta\}\}$;

7^0 *Stay-Shift:*

$(s_1, \pi_1, \#\alpha, \#u\,b\#, \beta\#, \pi_2, \lambda) \vdash (s_1, \pi_1, \alpha, \#u\#, b\,\beta\#, \pi_2, \lambda)$;

8^0 *Stay-Reduce:*

$(s_1, \pi_1, \#\alpha, \#u\#, \gamma\,\beta\#, \pi_2, \lambda) \vdash (s_1, \pi_1, \#\alpha, \#u\#, B_{b,e}\,\beta\#, r_2\pi_2, \lambda)$,
$r_2 = no(B \to h(\gamma))$, $e = |\pi_2| + 1$, $b = \min\{|\pi_2| + 1, \min\{b' \mid D_{b',e'} \in \gamma\}\}$;

9^0 *Possible-accept:*

$(s_1, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda) \vdash (s_2, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda)$;

10^0 *Shift-Terminal:*

$(s_2, \pi_1, \#\alpha, \#\#, a\,\beta\#, \pi_2, \pi_3) \vdash (s_2, \pi_1, \#\alpha\,a, \#\#, \beta\#, \pi_2, \pi_3)$;

11^0 *Shift-Nonterminal:*

$(s_2, \pi_1, \#\alpha, \#\#, A_{b,e}\,\beta\#, \pi_2'\pi_2'', \pi_3) \vdash (s_2, \pi_1, \#\alpha\,A, \#\#, \beta\#, \pi_2'', \pi_2'\pi_3)$, *where* $\pi_2' = [r_b, r_{b+1}, ..., r_e]$ *and* $|\pi_2'| = e - b + 1$;

12^0 *Reduce:*

$(s_2, \pi_1'\pi_1'', \#\alpha\,\beta, \#\#, \gamma\#, \pi_2, \pi_3) \vdash (s_2, \pi_1'', \#\alpha\,A, \#\#, \gamma\#, \pi_2, r_1\pi_3\pi_1')$,

where $r_1 = no(A \to h(\beta))$, $\beta = u_1\,B_{b,e}\,...\,u_m\,C_{b',e'}\,\beta'$, u_1, ..., $u_m \in V_T^*$, $|\pi_1'| = e' - b + 1$;

13^0 *Accept:* $(s_2, \lambda, \#S, \#\#, \#, \lambda, \pi) \vdash ACC$;

14^0 *Reject:* $(s_1, \pi_1, \#\alpha, \#u\#, \beta\#, \pi_2, \lambda) \vdash REJ$ *and also*

$(s_2, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \pi_3) \vdash REJ$ *if no transitions of type* 1^0, 2^0, ..., 13^0 *can be applied.*

The *deterministic two-stack machine*, which is a deterministic Turing machine with a read-only input and two storage tapes is known to have the same power as classical Turing machines ([HoU79]). If the heads moves left on either tape, a blank is printed on that tape. In [HoU79] the following lemma was proved: *An arbitrary single-tape Turing machine can be simulated by a deterministic two-stack machine.*

Another computational model equivalent to Turing machines is the two-counter machine, which is an off-line Turing machine whose storage is semi-infinite, and whose alphabet contains only two symbols, Z (the bottom of stack) and B (blank). 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 it cannot directly test whether two numbers are equal. *A two-counter machine can simulate an arbitrary Turing machine* ([HoU79]).

Our model is in fact a two-stack machine. The differences (between our model and the classical one) consist in the existence of two heads and two output tapes. The heads can simultaneously read the input tape and the two output tapes may be accessed only in write style. Only two states are used. Therefore, our model (Figure 5.2) can simulate a Turing machine.

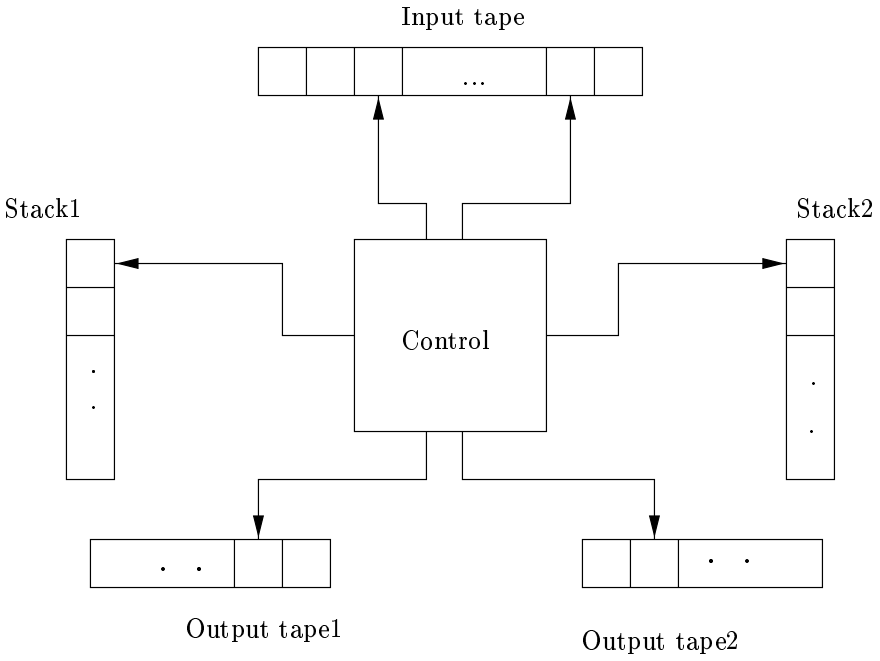


Figure 5.2. General Up-to-Up Bidirectional Parser Style

Lemma 5.1.1 *Let G be a context free grammar. If*

$$\textbf{(1)} \quad (s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{*}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda)$$

then $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$. (where h is presented in Notation 5.1.1)

Proof We proceed by induction on the number of transitions (denoted by t). Some notations: $\underset{G_u BP(G)}{\overset{t,*}{\vdash}}$ and $\underset{G_u BP(G)}{\overset{1^0}{\vdash}}$, mean that t transitions, respectively the transition 1^0 have been applied.
Basis: $t = 1$. Starting from the initial configuration, we can apply either of the transitions 1^0, 5^0 and 7^0. For 1^0, we obtain:

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{1^0}{\vdash}} (s_1, \lambda, \#\alpha, \#u_1' u_2 u_3'\#, \beta\#, \lambda, \lambda),$$

where $u_1 = a\, u_1'$ and $u_3 = u_3'\, b$. Of course, $a \underset{G,rm}{\overset{0}{\Longrightarrow}} a$ and $b \underset{G,rm}{\overset{0}{\Longrightarrow}} b$. The other cases ($5^0$ and 7^0) can be treated in a similar way.
Inductive Step: Suppose that the relation (1) is true for at most t transitions and prove it for $t+1$ transitions. We know that:

$$\textbf{(2)} \quad (s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{t+1,*}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda)$$

We have to prove that $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_2$. The last transition in (2) may be of one of the types 1^0, 2^0, ..., 8^0.

I: Suppose that the last transition in (2) is of the form *shift-shift*. We may rewrite (2) into:

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{t,*}{\vdash}} (s_1, \pi_1, \#\alpha', \#a\, u_2\, b\#, \beta'\#, \pi_2, \lambda)$$

$$\underset{G_u BP(G)}{\overset{1^0}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda), \text{ where } \alpha = \alpha'\, a,\ \beta = b\, \beta'.$$

According to the inductive hypothesis, we obtain:

$h(\alpha') \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1'$, $u_1 = u_1'\, a$ and $h(\beta') \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3'$, $u_3 = b\, u_3'$.

Therefore, $h(\alpha) = h(\alpha'\, a) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1'\, a$ and $h(\beta) = h(b\, \beta') \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} b\, u_3'$.

II: Suppose that the last transition in (2) is of the form *reduce-shift*. Rewrite (2) into:

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{t,*}{\vdash}} (s_1, \pi_1', \#\alpha'\, \beta', \#u_2\, b\#, \gamma\#, \pi_2, \lambda)$$

$$\underset{G_u BP(G)}{\overset{2^0}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda), \text{ where } r_1 = no(A \to h(\beta')),$$

$\alpha'\, A_{b',e'} = \alpha$, $e' = |\pi_1'| + 1$, $b' = \min\{|\pi_1'| + 1, \min\{b'' \mid C_{b'',e''} \in \beta'\}\}$, $b\, \gamma' = \beta$, $r_1 \pi_1' = \pi_1$. According to the inductive hypothesis, we get:

$h(\alpha'\, \beta') \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$, and $h(\gamma') \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3'$, $u_3 = b\, u_3'$.

We have:

$h(\alpha) = h(\alpha'\, A_{b',e'}) \underset{G,rm}{\overset{r_1}{\Longrightarrow}} h(\alpha'\, \beta') \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$, so $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and

$h(\beta) = h(b\, \gamma') = b\, h(\gamma') \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} b\, u_3' = u_3$, $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$.

III: Suppose that the last transition in (2) is of the form *shift-reduce*. Rewrite (2) into:

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \underset{G_u BP(G)}{\overset{t,*}{\vdash}} (s_1, \pi_1, \#\alpha', \#a\, u_2\#, \gamma\, \beta'\#, \pi_2', \lambda)$$

$$\underset{G_u BP(G)}{\overset{3^0}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda), \text{ where } \alpha'\, a = \alpha,\ r_2 = no(B \to h(\gamma)),$$

$\beta = B_{b,e}\, \beta'$, $e = |\pi_2'| + 1$, $b = \min\{|\pi_2'| + 1, \min\{b' \mid D_{b',e'} \in \gamma\}\}$, $r_2 \pi_2' = \pi_2$. Applying the inductive hypothesis, we get:

$h(\alpha') \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1'$, where $u_1 = u_1'\, a$, and $h(\gamma\, \beta') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$.

It follows that:

$h(\alpha) = h(\alpha'\, a) = h(\alpha')\, a \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1'\, a = u_1$, so $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and

$h(\beta) = h(B_{b,e}\, \beta') \underset{G,rm}{\overset{r_2}{\Longrightarrow}} h(\gamma\, \beta') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$, so $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$

IV: Suppose that the last transition in (2) is of the form *reduce-reduce*. Rewrite (2) as:

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \overset{t,*}{\underset{G_u BP(G)}{\vdash}} (s_1, \pi_1', \#\alpha'\, \beta', \#u_2\#, \varepsilon'\, \gamma'\#, \pi_2', \lambda)$$

$$\overset{4^0}{\underset{G_u BP(G)}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda), \text{ where } r_1 = no(A \to h(\beta')),$$

$\alpha'\, A_{b_1,e_1} = \alpha$, $r_1\pi_1' = \pi_1$, $e' = |\pi_1'|+1$, $b_1 = \min\{|\pi_1'|+1, \min\{b_1' \mid C_{b_1',e_1'} \in \beta'\}\}$, $r_2 = no(B \to h(\varepsilon'))$, $B_{b_2,e_2}\, \gamma' = \beta$, $r_2\pi_2' = \pi_2$, $e_2 = |\pi_2'| + 1$,
$b_2 = \min\{|\pi_2'|+1, \min\{b_2' \mid D_{b_2',e_2'} \in \varepsilon'\}\}$. According to the inductive hypothesis, we get:

$$h(\alpha'\, \beta') \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1, \text{ and } h(\varepsilon'\, \gamma') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3.$$

Now we have:

$h(\alpha) = h(\alpha'\, A_{b_1,e_1}) \underset{G,rm}{\overset{r_1}{\Longrightarrow}} h(\alpha'\, \beta') \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$, so $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and

$h(\beta) = h(B_{b_2,e_2}\, \gamma') \underset{G,rm}{\overset{r_2}{\Longrightarrow}} h(\varepsilon'\, \gamma') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$, so $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$.

Transitions 5^0, 6^0, 7^0, 8^0 can be treated in a similar way. ■

Lemma 5.1.2 *Let G be a context free grammar. If $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$ then*

$$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \overset{*}{\underset{G_u BP(G)}{\vdash}} (s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda)$$

Proof By induction on $t = |\pi_1| + |\pi_2|$.
Basis: $t = 0$.

This means $|\pi_1| = 0$ and $|\pi_2| = 0$. Then the hypothesis may be rewritten as $u_1 \underset{G,rm}{\overset{0}{\Longrightarrow}} u_1$ and $u_3 \underset{G,rm}{\overset{0}{\Longrightarrow}} u_3$, where $\alpha = u_1$ and $\beta = u_3$. Applying $\min\{|u_1|, |u_2|\}$ *shift-shift* transitions, and then $|u_1| - |u_3|$ *shift-stay* or $|u_3| - |u_1|$ *stay-shift* transitions, (depending on whether $|u_1| > |u_3|$ or not), consecutively, in this order, we get the required relation.

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

Case I: Let $r_1 = no(A \to \alpha_1)$ be the last applied production in $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$. Therefore, we have:

$$h(\alpha) = h(\alpha_2)\, A\, u_1' \underset{G,rm}{\overset{r_1}{\Longrightarrow}} h(\alpha_2)\, \alpha_1\, u_1' \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$$

Now, because $u_1 = u_1''\, u_1'$, we have to treat the derivation $h(\alpha_2\, \alpha_1) \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1''$. Applying the inductive hypothesis, we get:

$(s_1, \lambda, \#, \#u_1\, u_2\, u_3\#, \#, \lambda, \lambda) \overset{*}{\underset{G_u BP(G)}{\vdash}} (s_1, \pi_1', \#\alpha_2\, \alpha_1, \#u_1'\, u_2\#, \beta\#, \pi_2, \lambda)$

$\overset{6^0}{\underset{G_u BP(G)}{\vdash}} (s_1, r_1\pi_1', \#\alpha_2\, A_{b',e'}, \#u_1'\, u_2\#, \beta\#, \pi_2, \lambda),$

where $e' = |\pi_1'| + 1$, $b' = \min\{|\pi_1'| + 1, \min\{b'' \mid C_{b'',e''} \in \alpha_1\}\}$. Continuing with $|u_1'|$ transitions of type 5^0 ($\pi_1 = r_1\pi_1'$ and $h(\alpha_2\, A_{b',e'}\, u_1') = h(\alpha)$), we finally obtain the configuration $(s_1, \pi_1, \#\alpha, \#u_2\#, \beta\#, \pi_2, \lambda)$.

The case II can be solved analogously. ■

Lemma 5.1.3 *Let G be a context free grammar. Let* **(3)** *be*

$(s_2, \pi_1'\pi_1'', \#\alpha_1\alpha_2, \#\#, \beta_1\beta_2\#, \pi_2'\pi_2'', \lambda) \overset{*}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3),$

where $h(\alpha_2) \overset{\pi_1'}{\underset{G,rm}{\Longrightarrow}} u_1$ *and* $h(\beta_1) \overset{\pi_2'}{\underset{G,rm}{\Longrightarrow}} u_3$. *Then* $h(\gamma) \overset{\pi_3}{\underset{G,rm}{\Longrightarrow}} u_1\, u_3$.

Proof We proceed by induction on the number of (possible applied) transitions (denoted by t).

Basis: $t = 1$. Starting from the initial configuration the only applicable transitions are 10^0, 11^0 and 12^0.

(i) For 10^0, we obtain:

$(s_2, \pi_1'\pi_1'', \#\alpha, \#\#, a\,\beta\#, \pi_2'\pi_2'', \lambda) \underset{G_u BP(G)}{\vdash} (s_2, \pi_1'\pi_1'', \#\alpha\, a, \#\#, \beta\#, \pi_2'\pi_2'', \lambda).$

Obviously, we have $\alpha_1 = \alpha$, $\alpha_2 = \lambda$, $\beta_1 = \gamma = a$, $\beta_2 = \beta$, $\pi_1' = \lambda$, $\pi_2' = \lambda$, thus $h(\alpha_2) = \lambda \overset{0}{\underset{G,rm}{\Longrightarrow}} \lambda = u_1$ and $h(\beta_1) = \gamma = a \overset{0}{\underset{G,rm}{\Longrightarrow}} a = u_3$. Then $h(\gamma) \overset{0}{\underset{G,rm}{\Longrightarrow}} a = u_1\, u_3$.

(ii) For 11^0, we get:

$(s_2, \pi_1'\pi_1'', \#\alpha, \#\#, A_{b,e}\,\beta\#, \pi_2'\pi_2'', \lambda) \underset{G_u BP(G)}{\vdash} (s_2, \pi_1'\pi_1'', \#\alpha\, A, \#\#, \beta\#, \pi_2'', \pi_2'),$

where $|\pi_2'| = e - b + 1$. Obviously, we have $\alpha_1 = \alpha$, $\alpha_2 = \lambda$, $\beta_1 = A_{b,e}$, $\beta_2 = \beta$, $\gamma = A$, $\pi_1' = \lambda$. Thus $h(\alpha_2) = \lambda \overset{0}{\underset{G,rm}{\Longrightarrow}} \lambda = u_1$ and $h(\beta_1) = A \overset{\pi_2'}{\underset{G,rm}{\Longrightarrow}} u_3$. Then $h(\gamma) = A \overset{\pi_2'}{\underset{G,rm}{\Longrightarrow}} u_3 = u_1\, u_3$.

(iii) In a similar way, for 12^0 we obtain:

$(s_2, \pi_1'\pi_1'', \#\alpha_1\, \alpha_2, \#\#, \gamma\#, \pi_2, \lambda) \underset{G_u BP(G)}{\vdash} (s_2, \pi_1'', \#\alpha_1\, A, \#\#, \gamma\#, \pi_2, r_1\pi_1'),$

where $r_1 = no(A \to h(\alpha_2))$, $\alpha_2 = v_1\, B_{b,e} \ldots v_m\, C_{b',e'}\, \alpha_2'$, v_1, ..., $v_m \in V_T^*$ and $|\pi_1'| = e' - b + 1$. So, $\gamma = A$, $\beta_1 = \lambda$, $\pi_2' = \lambda$, $h(\alpha_2) = v_1\, B \ldots v_m\, C\, h(\alpha_2') \overset{\pi_1'}{\underset{G,lm}{\Longrightarrow}} u_1$ and $h(\beta_1) = \lambda \overset{0}{\underset{G,rm}{\Longrightarrow}} \lambda = u_3$. Then $h(\gamma) = A \overset{r_1\pi_1'}{\underset{G,lm}{\Longrightarrow}} u_1 = u_1\, u_3$.

Inductive Step: Suppose that relation (1) is true for at most t transitions and prove it for $t+1$ transitions. We know that (**4**):

$$(s_2, \pi_1'\pi_1'', \#\alpha_1\alpha_2, \#\#, \beta_1\beta_2\#, \pi_2'\pi_2'', \lambda) \overset{t+1,*}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3),$$

$h(\alpha_2) \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$ and $h(\beta_1) \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$. We have to prove that $h(\gamma) \underset{G,lm}{\overset{\pi_3}{\Longrightarrow}} u_1\, u_3$. The last transition in (4) may be of one of the types 10^0, 11^0 and 12^0.

I: Suppose that the last transition in (4) is of the form *shift-terminal*. This means that $\beta_1^{(1)} \in V_T$ (i.e. $\beta_1 = \beta_1'\, a$). Obviously, $h(\beta_1') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3'$ (where $u_3 = u_3'\, a$). Rewriting the transitions from (4), we obtain:

$$(s_2, \pi_1'\pi_1'', \#\alpha_1\, \alpha_2, \#\#, \beta_1\, \beta_2\#, \pi_2'\pi_2'', \lambda) \overset{t,*}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, \pi_1'', \#\alpha_1\gamma', \#\#, a\beta_2\#, \pi_2'', \pi_3') \overset{10^0}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3),$$ where

$\gamma = \gamma'\, a$. Because $h(\alpha_2) \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_1$ and $h(\beta_1') \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3'$, and applying the induction hypothesis, we have $h(\gamma') \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1\, u_3'$. Now, $h(\gamma) = h(\gamma'\, a) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1\, u_3'\, a = u_1\, u_3$.

II: Suppose that the last transition in (4) is of the form *shift-nonterminal*. Rewriting the transitions from (4), we get:

$$(s_2, \pi_1'\pi_1'', \#\alpha_1\, \alpha_2, \#\#, \beta_1'\, A_{b,e}\, \beta_2\#, \pi_{22}'\pi_{21}''\pi_2'', \lambda) \overset{t,*}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, \pi_1'', \#\alpha_1\gamma', \#\#, A_{b,e}\beta_2\#, \pi_{21}'\pi_2'', \pi_3') \overset{11^0}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3),$$

where $\gamma = \gamma'\, A$, $\beta_1 = \beta_1'\, A_{b,e}$, $\pi_3 = \pi_{21}'\pi_3'$, $|\pi_{21}'| = e - b + 1$, $(\pi_2' = \pi_{22}'\pi_{21}')$. We know that $h(\alpha_2) \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$ and $h(\beta_1) \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$. Because $h(\beta_1) \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$, it follows that $h(\beta_1') \underset{G,rm}{\overset{\pi_{22}'}{\Longrightarrow}} u_3'$ and $h(A) \underset{G,rm}{\overset{\pi_{21}'}{\Longrightarrow}} u_3''$, where $u_3'\, u_3'' = u_3$. Applying the inductive hypothesis, we have $h(\gamma') \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} u_1\, u_3'$. Now, we get

$$h(\gamma) = h(\gamma'\, A) \underset{G,rm}{\overset{\pi_{21}'}{\Longrightarrow}} h(\gamma')\, u_3'' \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} u_1\, u_3'\, u_3'' = u_1\, u_3$$

III: Suppose that the last transition in (4) is of the form *reduce*. Rewriting the transitions from (4), we get:

$$(s_2, \pi_{11}'\pi_{12}'\pi_1'', \#\alpha_1\, \alpha_{21}'\, \alpha_{23}', \#\#, \beta_1\, \beta_2\#, \pi_2'\pi_2'', \lambda) \overset{t,*}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, \pi_{12}'\pi_1'', \#\alpha_1\alpha_{21}'\alpha_{22}', \#\#, \beta_2\#, \pi_2'', \pi_3') \overset{12^0}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3),$$

$\gamma = A$, $r_1 = no(A \to h(\alpha_{21}'\, \alpha_{22}'))$, $\pi_3 = r_1\pi_3'\pi_{12}'$, $\alpha_{21}'\alpha_{22}' = v_1\, B_{b,e} \ldots v_m\, C_{b',e'}\, \beta'$, $v_1, \ldots, v_m \in V_T^*$, $|\pi_1'| = e' - b + 1$, $\pi_1' = \pi_{11}'\pi_{12}'$, $\alpha_2 = \alpha_{21}'\, \alpha_{23}'$.

We know that $h(\alpha_2) \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$, $h(\beta_1) \underset{G,rm}{\overset{\pi_2'}{\Longrightarrow}} u_3$ and we have to prove that $h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1\, u_3$. But $\alpha_2 = \alpha'_{21}\, \alpha'_{23}$, so $h(\alpha_2) = h(\alpha'_{21}) \cdot h(\alpha'_{23}) \underset{G,rm}{\overset{\pi_1'}{\Longrightarrow}} u_1$. Using the locality property for context free languages (Theorem 2.1.1), it follows that $h(\alpha'_{21}) \underset{G,rm}{\overset{\pi'_{12}}{\Longrightarrow}} u_{11}$, $h(\alpha'_{23}) \underset{G,rm}{\overset{\pi'_{11}}{\Longrightarrow}} u_{12}$ and $u_{11}\, u_{12} = u_1$.

From the inductive hypothesis we obtain that $h(\alpha'_{22}) \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} u_{12}\, u_3$. Then:

$h(\gamma) = A \underset{G,rm}{\overset{r_1}{\Longrightarrow}} h(\alpha'_2) = h(\alpha'_{21})h(\alpha'_{22}) \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} h(\alpha'_{21})u_{12}u_3 \underset{G,rm}{\overset{\pi'_{12}}{\Longrightarrow}} u_{11}u_{12}u_3 = u_1\, u_3$, i.e. $h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1\, u_3$. ■

Lemma 5.1.4 *Let $G = (V_N, V_T, S, P)$ be a context free grammar. If*

$A \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} h(v''_{n'} X_{b_{n',e_{n'}}} v_{n'+1} ... X_{b_{n-1},k} v_n)$, $A \in V_N$, $n' \leq n$ *and* $k = e_{n-1}$ *then*

$(s_2, [r_k, r_{k-1}, ..., r_1], \#v_1 X_{b_1,e_1} v_2 ... v_{n-1} X_{b_{n-1},k} v_n, \#\#, \beta_2\#, \pi''_2, \lambda) \overset{12^0,*}{\underset{G_u BP(G)}{\vdash}}$
$(s_2, [r_{k'}, .., r_1], \#v_1 X_{b_1,e_1} v_2 .. v_{n'-1} X_{b_{n'-1},k'} v'_{n'} A, \#\#, \beta_2\#, \pi''_2, \pi'_3 \cdot [r_k, .., r_{k'+1}])$,
where $v_{n'} = v'_{n'}\, v'_{n''}$ and $k' = e_{n'-1}$.

Proof We proceed by induction on $t = |\pi_3| > 0$.

Basis: $t = 1$. The initial derivation corresponds to a production in G, i.e. there exists the production $r = no(A \to h(v''_{n'} X_{b_{n',e_{n'}}} v_{n'+1} ... X_{b_{n-1},k} v_n))$. According to the transition 12^0, we obtain:

$(s_2, [r_k, r_{k-1}, ..., r_1], \#v_1 X_{b_1,e_1} v_2 ... v_{n-1} X_{b_{n-1},k} v_n, \#\#, \beta_2\#, \pi''_2, \lambda) \overset{12^0}{\underset{G_u BP(G)}{\vdash}}$
$(s_2, [r_{k'}, r_{k'-1}, .., r_1], \#v_1 X_{b_1,e_1} v_2 .. v_{n'-1} X_{b_{n'-1},k'} v'_n A, \#\#, \beta_2\#, \pi''_2, r \cdot [r_k, .., r_{k'+1}])$.

Inductive Step: $t > 1$. We denote $\pi'_3 = r\pi''_3$, where $r = no(A \to h(\beta))$. Therefore, our derivation could be rewritten into:

$$A \underset{G,rm}{\overset{r}{\Longrightarrow}} h(\beta) \underset{G,rm}{\overset{\pi''_3}{\Longrightarrow}} h(v''_{n'} X_{b'_n,e'_n} v_{n'+1} ... X_{b_{n-1},k} v_n)$$

Because $t > 1$, it follows that $|\pi''_3| \geq 1$. Due to the fact that we deal only with right most derivations, it results that $\beta^{(1)} \in V_N$ and $\beta = v''_{n'} X_{b'_n,e'_n} v_{n'+1} ... X_{b_{n''-1},e_{n''}} v'_{n''} B$. Then we have

$$B \underset{G,rm}{\overset{\pi''_3}{\Longrightarrow}} h(v''_{n''} X_{b_{n''+1},e_{n''+1}} v_{n''+1} ... X_{b_{n-1},k} v_n),$$

where $v_{n''} = v'_{n''}\, v''_{n''}$. According to the inductive hypothesis, we get:

$(s_2, [r_k, r_{k-1}, ..., r_1], \#v_1 X_{b_1,e_1} v_2 ... v_{n-1} X_{b_{n-1},k} v_n, \#\#, \beta_2\#, \pi''_2, \lambda) \overset{12^0,*}{\underset{G_u BP(G)}{\vdash}}$

$(s_2, [r_{k''}, r_{k''-1}, ..., r_1], \#v_1 X_{b_1,e_1} v_2 ... v_{n''-1} X_{b_{n''-1},k''} v'_{n''} B, \#\#, \beta_2\#, \ \pi''_2, \ \pi''_3 \cdot [r_k, ..., r_{k''+1}])$, where $k'' = e_{n''-1}$. This may be continued with a transition of type 12^0, we finally get the configuration:
$(s_2, [r_{k'}, r_{k'-1}, ..., r_1], \#v_1 X_{b_1,e_1} v_2 ... v_{n'-1} X_{b_{n'-1},k'} v'_{n'} A, \#\#, \beta_2\#, \pi''_2, r\pi''_3 \cdot [r_k, ..., r_{k''+1}] \cdot [r_{k''}, ..., r_{k'+1}])$, where $v_{n'} = v'_n \, v''_n$, $k' = e_{n'-1}$. ■

Lemma 5.1.5 *Let G be a context free grammar. If the following derivations hold*

$h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1 \, u_3$, $h(\alpha_2) \underset{G,rm}{\overset{\pi'_1}{\Longrightarrow}} u_1$, $h(\beta_1) \underset{G,rm}{\overset{\pi'_2}{\Longrightarrow}} u_3$, *(and* $h(\gamma) \underset{G,rm}{\overset{*}{\Longrightarrow}} h(\alpha_2 \, \beta_1)$*)*

then (**5**)

$$(s_2, \pi'_1\pi''_1, \#\alpha_1 \, \alpha_2, \#\#, \beta_1 \, \beta_2\#, \pi'_2\pi''_2, \lambda) \overset{*}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi''_1, \#\alpha_1 \, \gamma, \#\#, \beta_2\#, \pi''_2, \pi_3)$$

Proof By induction on $t = |\beta_1|$.

Basis: $t = 0$. Thus $\beta_1 = \lambda$, $u_3 = \lambda$ and $\pi'_2 = \lambda$. We have $h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1$, $h(\alpha_2) \underset{G,rm}{\overset{\pi'_1}{\Longrightarrow}} u_1$, (and $h(\gamma) \underset{G,rm}{\overset{*}{\Longrightarrow}} h(\alpha_2)$). Therefore, starting from the initial configuration from (5), the $G_u BP(G)$ could make only transitions of type 12^0 and then $\gamma \in V_N$. We can consider the right most derivation: $\gamma = A_0 \underset{G,rm}{\overset{r_1}{\Longrightarrow}} \gamma_1 \, A_1 \underset{G,rm}{\overset{r_2}{\Longrightarrow}} \gamma_1 \, \gamma_2 \, A_2 ... \underset{G,rm}{\overset{r_n}{\Longrightarrow}} \gamma_1 \, \gamma_2 \, ... \, \gamma_n \, A_n = \alpha_2$. Then $\pi_3 = [r_1, r_2, ..., r_n] \cdot \pi_1$ and according to Lemma 5.1.4 we obtain (5).

Inductive Step: Suppose that the relation (5) is true for t, and prove for it $t + 1$. We distinguish two cases:

I: $\beta_1 = \beta'_1 \, a$. Due to $h(\beta_1) \underset{G,rm}{\overset{\pi'_2}{\Longrightarrow}} u_3$, it follows that $h(\beta'_1) \underset{G,rm}{\overset{\pi'_2}{\Longrightarrow}} u'_3$, where $u'_3 \, a = u_3$. Because $h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1 \, u'_3 \, a$, denoting the word γ' so that $\gamma = \gamma' \, a$, it follows that $h(\gamma') \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1 \, u'_3$ (and $h(\gamma') \underset{G,rm}{\overset{*}{\Longrightarrow}} h(\alpha_2 \, \beta'_1)$). Applying the inductive hypothesis it results that

$$(s_2, \pi'_1\pi''_1, \#\alpha_1\alpha_2, \#\#, \beta_1\beta_2\#, \pi'_2\pi''_2, \lambda) \overset{*}{\underset{G_u BP(G)}{\vdash}} (s_2, \pi''_1, \#\alpha_1\gamma', \#\#, a\beta_2\#, \pi''_2, \pi_3)$$

Now, we may apply transition 10^0 and finally obtain the configuration

$$(s_2, \pi''_1, \#\alpha_1 \, \gamma, \#\#, \beta_2\#, \pi''_2, \pi_3).$$

II: $\beta_1 = \beta'_1 \, A_{b,e}$. We have $h(\beta'_1 \, A_{b,e}) \underset{G,rm}{\overset{\pi'_2}{\Longrightarrow}} u_3$. Let π'''_2 be the right most derivation ($|\pi'''_2| = e - b + 1$) for which $h(A_{b,e}) \underset{G,rm}{\overset{\pi'''_2}{\Longrightarrow}} u''_3$. Let us denote by π'_3 the right most derivation for which $h(\beta'_1) \underset{G,rm}{\overset{\pi'_3}{\Longrightarrow}} u'_3$. Obviously, $u_3 = u'_3 \, u''_3$ (because $h(\gamma) \underset{G,rm}{\overset{*}{\Longrightarrow}} h(\alpha_2 \, \beta_1)$).

Let us denote γ' such that $\gamma = \gamma' A_{b,e}$. Because $h(A_{b,e}) \underset{G,rm}{\overset{\pi_2'''}{\Longrightarrow}} u_3''$ and $h(\gamma) \underset{G,rm}{\overset{\pi_3}{\Longrightarrow}} u_1 u_3' u_3''$, it follows that $h(\gamma') \underset{G,rm}{\overset{\pi_3'}{\Longrightarrow}} u_1 u_3'$ and $\pi_3 = \pi_2''' \pi_3'$.

According to the inductive hypothesis:

$(s_2, \pi_1'\pi_1'', \#\alpha_1\alpha_2, \#\#, \beta_1\beta_2\#, \pi_2'\pi_2'', \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}}$

$\overset{*}{\underset{G_uBP(G)}{\vdash}} (s_2, \pi_1'', \#\alpha_1\gamma', \#\#, A_{b,e}\beta_2\#, \pi_2'''\pi_2'', \pi_3')$

Now, we may apply the transition 11^0 and finally obtain the configuration

$$(s_2, \pi_1'', \#\alpha_1\,\gamma, \#\#, \beta_2\#, \pi_2'', \pi_3).$$

■

We present below the main result which ensures the correctness of $G_uBP(G)$.

Theorem 5.1.1 *Let $G = (V_N, V_T, S, P)$ be a context free grammar. Then*

(6) $(s_1, \lambda, \#, \#w\#, \#, \lambda, \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}} (s_2, \lambda, \#S, \#\#, \#, \lambda, \pi) \overset{13^0}{\underset{G_uBP(G)}{\vdash}} ACC$

iff $S \underset{G,rm}{\overset{\pi}{\Longrightarrow}} w$.

Proof

($\Longrightarrow$) Following transition 9^0, the transition (6) could be rewritten as:

$$(s_1, \lambda, \#, \#w\#, \#, \lambda, \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}} (s_1, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda) \overset{9^0}{\underset{G_uBP(G)}{\vdash}}$$

$$(s_2, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}} (s_2, \lambda, \#S, \#\#, \#, \lambda, \pi) \overset{13^0}{\underset{G_uBP(G)}{\vdash}} ACC$$

According to Lemma 5.1.1 ($u_2 = \lambda$ and $u_1\,u_3 = w$) we obtain $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$. Now, according to Lemma 5.1.3, and taking $\alpha_1 = \beta_2 = \lambda$, $\gamma = S$, $\pi_3 = \pi$, we obtain $S \underset{G,rm}{\overset{\pi}{\Longrightarrow}} u_1\,u_3 = w$.

($\Longleftarrow$) We know that $S \underset{G,rm}{\overset{\pi}{\Longrightarrow}} w$. So there exist u_1, $u_3 \in V_T^*$, $\alpha, \beta \in V'^*$, π_1, π_2' (right most derivations) for which $u_1\,u_3 = w$, $h(\alpha) \underset{G,rm}{\overset{\pi_1}{\Longrightarrow}} u_1$ and $h(\beta) \underset{G,rm}{\overset{\pi_2}{\Longrightarrow}} u_3$. According to Lemma 5.1.2, and taking $u_2 = \lambda$, it results that

$$(s_1, \lambda, \#, \#w\#, \#, \lambda, \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}} (s_1, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda).$$

From Lemma 5.1.5 and taking $\alpha_1 = \beta_1 = \lambda$, $\pi_1'' = \pi_2'' = \lambda$, it follows that

$(s_2, \pi_1, \#\alpha, \#\#, \beta\#, \pi_2, \lambda) \overset{*}{\underset{G_uBP(G)}{\vdash}} (s_2, \lambda, \#S, \#\#, \#, \lambda, \pi) \overset{13^0}{\underset{G_uBP(G)}{\vdash}} ACC.$ ■

Example 5.1.1 *Let $G = (\{A, B, C\}, \{a, b, c, d, e\}, A, P)$ be a context free grammar, where the set of productions P is given by:*

1. $A \to a\,B\,A\,C\,b$ **2.** $A \to d\,A$ **3.** $A \to e$ **4.** $B \to b\,B\,c$
5. $B \to d$ **6.** $C \to c\,C$ **7.** $C \to d$

Let $w = a\,d\,a\,b\,d\,c\,d\,e\,d\,b\,c\,c\,d\,b$ be the input word for the parser $G_uBP(G)$. A possible sequence of transitions performed by $G_uBP(G)$ could be:

$(s_1, \lambda, \#, \#a\,d\,a\,b\,d\,c\,d\,e\,d\,b\,c\,c\,d\,b\#, \#, \lambda, \lambda) \overset{1^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, \lambda, \#a, \#d\,a\,b\,d\,c\,d\,e\,d\,b\,c\,c\,d\#, b\#, \lambda, \lambda) \overset{1^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, \lambda, \#a\,d, \#a\,b\,d\,c\,d\,e\,d\,b\,c\,c\#, d\,b\#, \lambda, \lambda) \overset{4^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5], \#a\,B_{1,1}, \#a\,b\,d\,c\,d\,e\,d\,b\,c\,c\#, C_{1,1}\,b\#, [7], \lambda) \overset{1^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5], \#a\,B_{1,1}\,a, \#b\,d\,c\,d\,e\,d\,b\,c\#, c\,C_{1,1}\,b\#, [7], \lambda) \overset{3^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5], \#a\,B_{1,1}\,a\,b, \#d\,c\,d\,e\,d\,b\,c\#, C_{1,2}\,b\#, [6, 7], \lambda) \overset{1^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5], \#a\,B_{1,1}\,a\,b\,d, \#c\,d\,e\,d\,b\#, c\,C_{1,2}\,b\#, [6, 7], \lambda) \overset{4^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5, 5], \#a\,B_{1,1}\,a\,b\,B_{2,2}, \#c\,d\,e\,d\,b\#, C_{1,3}\,b\#, [6, 6, 7], \lambda) \overset{1^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [5, 5], \#a\,B_{1,1}\,a\,b\,B_{2,2}\,c, \#d\,e\,d\#, b\,C_{1,3}\,b\#, [6, 6, 7], \lambda) \overset{2^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}, \#d\,e\#, d\,b\,C_{1,3}\,b\#, [6, 6, 7], \lambda) \overset{3^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}\,e, \#d\#, C_{4,4}\,b\,C_{1,3}\,b\#, [7, 6, 6, 7], \lambda) \overset{5^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_1, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}\,d\,e, \#\#, C_{4,4}\,b\,C_{1,3}\,b\#, [7, 6, 6, 7], \lambda) \overset{9^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_2, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}\,d\,e, \#\#, C_{4,4}\,b\,C_{1,3}\,b\#, [7, 6, 6, 7], \lambda) \overset{12^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_2, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}\,d\,A, \#\#, C_{4,4}\,b\,C_{1,3}\,b\#, [7, 6, 6, 7], [3]) \overset{12^0}{\underset{G_uBP(G)}{\vdash}}$

$(s_2, [4, 5, 5], \#a\,B_{1,1}\,a\,B_{2,3}\,A, \#\#, C_{4,4}\,b\,C_{1,3}\,b\#, [7, 6, 6, 7], [2, 3]) \overset{11^0}{\underset{G_uBP(G)}{\vdash}}$

$$(s_2, [4,5,5], \#a\,B_{1,1}\,a\,B_{2,3}\,A\,C, \#\#, b\,C_{1,3}\,b\#, [6,6,7], [7,2,3]) \overset{10^0}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, [4,5,5], \#a\,B_{1,1}\,a\,B_{2,3}\,A\,C\,b, \#\#, C_{1,3}\,b\#, [6,6,7], [7,2,3]) \overset{12^0}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, [5], \#a\,B_{1,1}\,A, \#\#, C_{1,3}\,b\#, [6,6,7], [1,7,2,3,4,5]) \overset{11^0}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, [5], \#a\,B_{1,1}\,A\,C, \#\#, b\#, \lambda, [6,6,7,1,7,2,3,4,5]) \overset{10^0}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, [5], \#a\,B_{1,1}\,A\,C\,b, \#\#, \#, \lambda, [6,6,7,1,7,2,3,4,5]) \overset{12^0}{\underset{G_u BP(G)}{\vdash}}$$

$$(s_2, \lambda, \#A, \#\#, \#, \lambda, [1,6,6,7,1,7,2,3,4,5,5]) \overset{13^0}{\underset{G_u BP(G)}{\vdash}} ACC$$

Therefore, w is accepted by $G_u BP(G)$, and the right most derivation is

$\pi_{rm} = [1,6,6,7,1,7,2,3,4,5,5]$.

5.2 Parallel approach for general up-to-up bidirectional parsing

In this section, we present a parallel approach convenient for describing the general up-to-up bidirectional parsing strategy.

Our model is a SIMD (simple instruction stream, multiple data stream) computer. We consider two processors `P1` and `P2` which operate simultaneously and synchronously. Furthermore, our model shares a common memory. The notions "*running time*" $t(n)$, "*computational steps*", "*routing steps*", "*speedup*", "*cost*", "*efficiency*" for parallel algorithms are presented in Section 2.3.2.

We shall present a parallel algorithm which describes the general up-to-up bidirectional parsing strategy. In fact, our parallel algorithm (denoted by (PAR-UUBP)) may be clearly derived from Definition 5.1.1 (general up-to-up bidirectional parser).

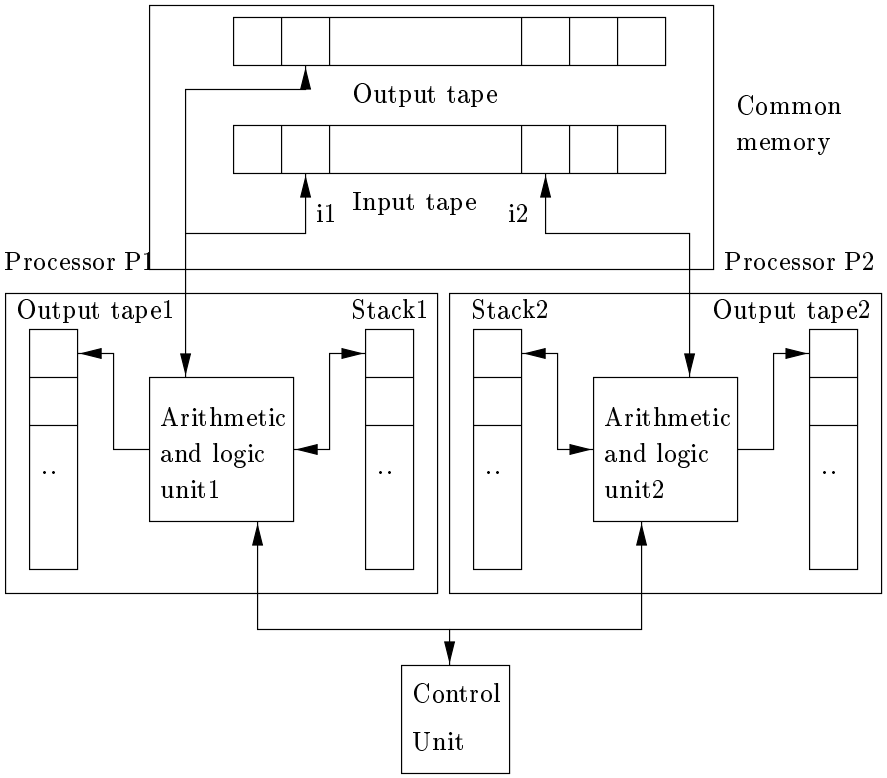


Figure 5.3. General SIMD Model for Up-to-Up Bidirectional Parsing

Our algorithm uses the following variables:

- `w`$\in V_T^*$ contains the input word (stored in the common memory);
- `n`$= |$`w`$|$;
- `i1`, `i2` are two counters used for indicating the positions of the pointers to `w` (stored in the common memory);
- `accept` is a boolean variable which takes the value true iff `w`$\in L(G)$ (stored in the common memory);
- `Stack1`, `Stack2` are two working stacks for `P1` and `P2`;
- `Output_tape1`, `Output_tape2` are the output tapes of `P1` and `P2` for storing the partial syntactic analysis of `w`;
- `Output_tape` is the output tape for storing the global syntactic analysis of `w` (stored in the common memory);
- `exit` is a boolean variable which is true iff `P1` or `P2` detect the non-acceptance of `w` (stored in the common memory).

We have also use some predefined procedures, such as:

- `pop(Stack,`α`)` - the resulting value of α will be the string of length $|\alpha|$ starting from the first symbol of `Stack`; after that, the string α is removed from `Stack`;
- `push_first(Stack,A)` - adds the symbol `A` to the content of `Stack`; `A` will be the new top of `Stack`;
- `push_last(Stack,`α`)` - adds the string α to the content of `Stack`, starting from the last symbol of `Stack`; `Stack` will have the same top.

The method of parallel algorithm (PAR-UUBP) can now be pointed out. Let the context free grammar $G = (V_N, V_T, S, P)$ be given. Then:

```
begin
  read(n); read(w); i1 := 1; i2 := n;
  accept := false; exit := false;
  repeat in parallel
    action1(P1);
    action2(P2)
  until (i1 > i2) or (exit = true);
  if (exit = true) then accept := false
  else
    repeat
      action3(P1,P2)
    until (exit = true);
  if (accept = true) then begin
    write('w is accepted and has right hand syntactical analysis');
    write(Output_tape)
  end
  else write('w is not accepted');
end.
```

It remains to describe the procedures `action1(P1)`, `action2(P2)` and `action3(P1,P2)`.

```
procedure action1(P1);
begin
   case
     if (∃ r1 =no( A→ h(α)), α belongs to Stack1 starting from the top)
     then begin
       /* reduce action */
       let α := u1 C_{b,e}...um D_{b',e'} α', where u1,...,um ∈ V_T^*;
       e'' := |Output_tape1| + 1;
       if (α does not contain any symbol from V' − V) then b'' := e''
       else b'' := min{e'', b};
```

```
      pop(Stack1,α);
      push_first(Stack1,A_{b'',e'});
      push_first(Output_tape1, r1);
    end;
    if (i1 <= i2) then begin
      /* shift action */
      push_first(Stack1,w[i1]);
      i1 := i1 + 1;
    end
    otherwise: begin
      backtracking step is needed;
      if (all the backtracking steps were terminated) and
         (no reduce or shift action could be performed)
      then exit := true;
    end
end;
```

The description of procedure `action2(P2)` is similar to `action1(P1)`.

```
procedure action2(P2);
begin
   case
    if (∃ r2 =no( A→ h(β)), β belongs to Stack1 starting from the top)
    then begin
      /* reduce action */
      let β := u_1 C_{b,e}...u_m D_{b',e'} β', where u_1, ..., u_m ∈ V_T^*;
      e'' := |Output_tape2| + 1;
      if (β does not contain any symbol from V' − V) then b'' := e''
      else b'' := min{e'', b};
      pop(Stack2,β);
      push_first(Stack2,B_{b'',e'});
      push_first(Output_tape2, r2);
    end;
    if (i1 < i2) then begin
      /* shift action */
      push_first(Stack2,w[i2]);
      i2 := i2 - 1;
    end
    otherwise: begin
      backtracking step is needed;
      if (all the backtracking steps are over) and
         (no reduce or shift action could be performed)
      then exit := true;
    end
end;
```

Finally, we describe the procedure `action3(P1,P2)` in a sequential way. The goal is to simulate the transitions 10^0, 11^0, 12^0 and 13^0 for $G_u BP(G)$ from Definition 5.1.1. The input tape is now empty, i.e. `w` has been already read (of course, if `exit` has the value $false$). Next we read symbols from `Stack2` (send by processor `P2`) modifying the content of the `Output_tape1` and `Output_tape2` putting the results in `Output_tape`.

```
procedure action3(P1,P2);
begin
   case
     if (∃ r1 = no( A→ h(α)), α is in Stack1 starting from the top)
     then begin
       /* reduce action */
       let α = u1 C_{b,e} ... α' D_{b',e'} α'', where u ∈ V_T^*, α'' ∈ V^*, α' ∈ (V ∪ V')^*;
       let π'_1 from Output_tape1 such that |π'_1| = e' − b + 1;
       pop(Output_tape1,π'_1);
       pop(Stack1,α);
       push(Stack2,A);
       push_first(Output_tape,r1);
       push_last(Output_tape,π'_1);
     end;
     if (top of Stack2 is a terminal symbol) then begin
       /* shift-terminal action */
       pop(Stack2,a), where a∈ V_T;
       push_first(Stack1,a);
     end;
     if (top of Stack2 is from V') then begin
       /* shift-nonterminal action */
       pop(Stack2,A_{b,e});
       push_first(Stack1,A);
       let π'_2 from Output_tape2 such that |π'_2| = e − b + 1;
       pop(Output_tape2,π'_2);
       push_first(Output_tape,π'_2);
     end;
     if (Output_tape1 = ∅) and (Output_tape2 = ∅) and
       (Stack1 = S) and (Stack2 = ∅) then begin
         accept := true; exit := true
     end;
   otherwise: begin
     backtracking step is needed;
     if (all the backtracking steps were terminated) and
       (no reduce or shift action could be performed)
     then exit := true
   end;
end;
```

Theorem 5.2.1 *(termination of Algorithm (PAR-UUBP))*
The Algorithm (PAR-UUBP) 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 *reduce actions* for processors `P1` and `P2` is finite. For the *shift actions* we read exactly one character from the input word `w`. Thus, because `w` is a finite word, it follows that the statement `repeat in parallel ... until` from the main program of Algorithm (PAR-UUBP) has a finite number of iterations (the execution of procedures `action1(P1)` and `action2(P2)` has a finite number of iterations).

The size of work-stacks is finite because `w` and G are finite. The other statement `repeat ... until` of the main program refers to `action3(P1,P2)`. The number of *shift actions* is finite due to the finiteness of `Stack2`, and the number of *reduce actions* is finite due to the finite number of productions of G.

Therefore, Algorithm (PAR-UUBP) performs a finite number of steps until terminates its execution. ■

Theorem 5.2.2 *(correctness of Algorithm (PAR-UUBP))*
For any given context free grammar $G = (V_N, V_T, S, P)$ and any $w \in V_T^*$, Algorithm (PAR-UUBP) gives the answer "`w is accepted and has right hand syntactical analysis` π*" if $S \underset{G,rm}{\overset{\pi}{\Longrightarrow}} w$ and "*`w is not accepted.`*" otherwise.*

Proof To show the correctness of Algorithm (PAR-UUBP), it suffices to remark that (PAR-UUBP) is "*equivalent*" to the sequential algorithm associated to the transitions of the general up-to-up bidirectional parser (Definition 5.1.1). In fact, the procedures `action1(P1)` and `action2(P2)` correspond to the transitions $1^0 - 8^0$ and `action3(P1,P2)` to the transitions $10^0 - 13^0$. Obviously, the transition 9^0 is realized by the main program of (PAR-UUBP).

The only situations for the parallel model in which the processors `P1` and `P2` work or wait (simultaneously) are:

(i) `P1` works, `P2` works

(ii) `P1` waits, `P2` works

(iii) `P1` works, `P2` waits

It is obvious that situation (i) (where `action1(P1)` and `action2(P2)` will be called and executed in parallel) is described in an equivalent way in the general up-to-up bidirectional parser by transitions $1^0 - 4^0$ and $9^0, 10^0$ (according to Definition 5.1.1).

Situation (ii) corresponds to transitions $7^0, 8^0$ and situation (iii) corresponds to transitions $5^0, 6^0, 12^0, 13^0$.

The nondeterministic behavior of the general up-to-up bidirectional parser is performed in the parallel algorithm by introducing these backtracking points for `action1(P1)`, `action2(P2)` and `action3(P1,P2)`. If no backtracking step

could be performed, then the variable `exit` is true. Then the input word is not accepted by the parser $G_u BP(G)$ (and of course, it is not in the language of the input grammar).

The parallel algorithm (PAR-UUBP) is then correct. ■

5.3 Deterministic up-to-up bidirectional parsing for context free languages

Deterministic (and linear) parallel algorithms (viewed as particular cases of the general up-to-up bidirectional parsing algorithm) for solving the membership problem, can be derived for some "combinations" of subclasses of context free languages. The deterministic up-to-up bidirectional parser has the same "device" as the general model. The only difference concerns the uniqueness of choosing the production `r` from a set of given productions of the input grammar (no backtracking steps are needed).

We have already presented the notions of $LR(k)$ ([Knu65]) and $RL(k)$ ([AnK99a]) grammars, $k \in \mathbf{N}$ (Definitions 2.1.15 and 4.3.7).

In a similar way, we can define the "*mirror*" of some classical subclasses of $LR(k)$ grammars.

Definition 5.3.1 *Let G be a context free grammar.*

- *G is a $SRL(1)$ (respectively $LARL(1)$) grammar if $\widetilde{G}$ is a $SLR(1)$ (respectively $LALR(1)$) grammar;*
- *L is a $SRL(1)$ (respectively $LARL(1)$) language if there exists a $SRL(1)$ grammar (respectively a $LARL(1)$) grammar which generates L.*

Now, we shall "combine" the $LR(k)$ and $RL(k)$ styles in order to obtain the deterministic up-to-up bidirectional parsing for context free languages.

Definition 5.3.2 *Let G be a context free grammar and k_1, $k_2 \in \mathbf{N}$. We say that G is a $LR(k_1) - RL(k_2)$ grammar iff G is both an $LR(k_1)$ and an $RL(k_2)$ grammar. A language L is called $LR(k_1) - RL(k_2)$ if there exists G, an $LR(k_1) - RL(k_2)$ grammar for which $L = L(G)$.*

The class of $LR(k)$ languages ($k \geq 1$) is the same as to the class of $LR(1)$ languages ([Knu65]). Then the above definition has a practical interest for k_1, $k_2 \in \{0, 1\}$ and for some subclasses of $LR(1)$ grammars (such as, $SLR(1)$ and $LALR(1)$). According to Definitions 5.3.1 and 5.3.2, we can easily derive all the **16** combinations between $LR(0)$, $SLR(1)$, $LALR(1)$, $LR(1)$ and $RL(0)$, $SRL(1)$, $LARL(1)$, $RL(1)$. For instance, G is a $LR(0) - LARL(1)$ grammar iff G is a $LR(0)$ and $LARL(1)$ grammar (Example 5.1.1).

We can also easily derive the relation between G and $\widetilde{G}$ for all the 16 combinations. For example, G is a $LR(0) - LARL(1)$ grammar iff $\widetilde{G}$ is a $LALR(1) - RL(0)$ grammar. The proof is obvious (Definitions 4.3.7, 5.3.1 and 5.3.2).

According to the classical hierarchies for the subclasses of $LR(1)$ languages, we can define some new hierarchies:

$$LR(0) \subset SLR(1) \subset LR(1) \text{ and } LR(0) \subset LALR(1) \subset LR(1)$$

Note that:

$$RL(0) \subset SRL(1) \subset RL(1) \text{ and } RL(0) \subset LARL(1) \subset RL(1)$$

These inclusions can be extended according to Definition 5.3.2 so that the following (16) inclusions hold:

$$LR(0) - RL(0) \subset SLR(1) - RL(0) \subset LR(1) - RL(0)$$

$$LR(0) - RL(0) \subset LR(0) - SRL(0) \subset LR(0) - RL(1)$$

$$LR(0) - RL(0) \subset SLR(1) - RL(0) \subset SLR(1) - SRL(1)$$

. . . . .

We have introduced and studied some new subclasses of languages larger that in Chapter 4 ([AnK99a]). For instance, the subclass $LL(k) - RL(1)$ defined in Chapter 5 (and [AnK99a]) is included in the subclass $LR(1) - RL(1)$ (Definition 5.3.2).

The deterministic up-to-up bidirectional parsing is similar to the general parallel approach and, the only difference is the absence of the backtracking steps. The procedures `action1(P1)` and `action2(P2)` are related to the classical sequential syntactical analysis algorithms for $LR(1)$ (or for the subclasses $LR(0)$, $SLR(1)$ and $LALR(1)$) and $RL(1)$ (or for the subclasses $RL(0)$, $SRL(1)$, $LARL(1)$). We also get a linear running time for the procedures `action1(P1)` and `action2(P2)` instead of an exponential sequential running time. Obviously, the procedure `action3(P1,P2)` has no backtracking steps for the subclasses of grammars given in Definition 5.3.2. Consequently, the procedure `action3(P1,P2)` is deterministic and has a linear running time.

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 analyzers for the mentioned subclasses of context free grammars ($LR(0)$, $SLR(1)$, $LALR(1)$, $LR(1)$, $RL(0)$, $SRL(1)$, $LARL(1)$, $RL(1)$).

Theorem 5.3.1 *(the complexity of the deterministic parallel algorithms)*
Let us denote respectively $T_1(n)$, $T_2(n)$ and $T_3(n)$ the running times of the sequential procedures `action1(P1)`, `action2(P2)` *and* `action3(P1,P2)`, *where n is the length of the input word. Suppose that the routing time is zero. Then the parallel running time $t(n)$ satisfies the relations:*

- $\frac{\min\{T_1(n),T_2(n)\}}{2} + T_3(n) \leq t(n) \leq \max\{T_1(n), T_2(n)\} + T_3(n)$;
- $t(n) \in \mathcal{O}(n)$.

Proof

(i) The inequality $t(n) \leq \max\{T_1(n), T_2(n)\}$ can be obtained by supposing that one processor waits. For instance, if `P2` waits, then $t(n) = T_1(n) + T_3(n)$ (time routing is zero).

The other inequality can be obtained by supposing that both processors work until $i1 = i2$. This means a running time equal to $\frac{\min\{T_1(n),T_2(n)\}}{2}$. Then processor `P2` will send to processor `P1` (when `P1` has to perform a *shift action*) a terminal or nonterminal symbol. Therefore $\frac{\min\{T_1(n),T_2(n)\}}{2} + T_3(n) \leq t(n)$.

(ii) Obviously, $T_1(n) \in \mathcal{O}(n)$ and $T_2(n) \in \mathcal{O}(n)$ following some classical results related to subclasses of context free grammars ($LR(0)$, $RL(0)$, $SLR(1)$, $SRL(1)$, ...). The local memories of `P1` and `P2` satisfy

- $|$`Stack1`$| \in \mathcal{O}(n)$, $|$`Stack2`$| \in \mathcal{O}(n)$, and
- $|$`Output_tape1`$| \in \mathcal{O}(n)$, $|$`Output_tape2`$| \in \mathcal{O}(n)$.

At every step of the parallel algorithm the size of the stacks (`Stack1`, `Stack2`, `Output_tape1` and `Output_tape2`) decrease. Furthermore, the procedure `action3(P1,P2)` is deterministic. Therefore, it follows that $T_3(n) \in \mathcal{O}(n)$. According to (i), it follows that $t(n) \in \mathcal{O}(n)$. ■

Considering the context free grammar from Example 5.1.1, which is $LR(0) - LARL(1)$, we see that the presented transitions correspond to a possible deterministic parallel running using our algorithm. Hence, a new example is not needed.

5.4 Conclusions

Our up-to-up bidirectional parser model is in fact a two-stack machine ([HoU79]) and therefore it can simulate a Turing machine. An interesting example of a bidirectional parser which can analyst a context sensitive language was presented in Example 4.1.2 and in [AnK99a]. The same may hold for the up-to-up bidirectional parser.

Compared to Chapter 4, where $LL(k)$, $LR(k)$ and precedence grammars styles have been combined, in Chapter 5 the bidirectional parsing uses only $LR(k)$ style (and its "*mirror*" $RL(k)$ style). The classes of languages described in this chapter are larger than the classes of languages defined in Chapter 4 (see also [AnK99a]).

Open problems:

- get a more precise estimation of the running time of the deterministic parallel algorithm presented in Section 5.3;
- find further closure properties (or local hierarchies) for subclasses of the described languages.

Chapter 6

Bidirectional attribute evaluation

In this chapter we describe a parallel algorithm (using two processors) for evaluating the attribute instances of an attributed derivation tree. It was prepared using the papers [AKM99] and [And97].

Section 6.1 emphasizes two ways for representing the (ordered) oriented trees. The specific bidirectional traversal is also pointed out.

Section 6.2 contains a new approach for evaluating the attribute instances of an attributed derivation tree. We have called this strategy *the bidirectional attribute evaluation.*

6.1 Data representations of trees and their bidirectional traversal

In this section, we shall present two methods for representing the ordered oriented trees. For the second one, a bidirectional traversal is presented.

First method of representation of ordered oriented trees.

Given the ordered oriented tree $T = (\mathcal{V}, E, s, d)$, let m be the maximal number of local sons (taking into account all vertices). For representing the sons $v_1, v_2, ..., v_n$ of a father v, we shall use exactly m locations (even if $n < m$). Consider now the array $t : \{1, 2, ..., p\} \to \mathcal{V} \cup \{null\}$, where p is a natural number (which will be defined later). The array t (denoted by `t[]`) is defined in the following way:

`t[]` $= root\ v_1 ... v_2\ v_{11} ... v_{1m}\ v_{21} ... v_{22} \ ...\ v_{1m} ... v_{mm} ... v\underbrace{mm...m}_{m\ times}$

constructed by structural induction:

• if the root of T has the sons $v_1, ..., v_n$ then `t[1]` $= root$, `t[2]` $= v_1$, ..., `t[n]` $= v_n$, `t[n+1]` $= null$, ..., `t[m]` $= null$, where $null$ is a special symbol;

• let $v_{wk} \in V(T)$ have exactly the sons v_{wk1}, v_{wk2}, ..., v_{wkn} and let `t[s]`$= v_{wk}$, where $w \in \{1,2,...,m\}^*$, $k \in \{1,2,...,m\}$. Then `t[s+(m-k)(m+1)+1]`$= v_{wk1}$, `t[s+(m-k)(m+1)+2]`$= v_{wk2}$, ..., `t[s+(m-k)(m+1)+n]`$= v_{wkn}$, and the "null" elements are `t[s+(m-k)(m+1)+n+1]`$= null$, ..., `t[s+(m-k)(m+1)+m]`$= null$.

This representation of an ordered oriented trees is especially useful for *breadth first search* visits. We do not present in detail this method of traversing ordered oriented trees, because this representation has a disadvantage related to its size. That is, the number of elements of the array `t[]` is exponential in m. In fact, $p = 1 + m + m^2 + ... + m^m = \frac{m^{m+1}-1}{m-1}$ $(m \neq 1)$, although the number of vertices of T could be "very" smaller when compared to this number.

The second method to represent ordered oriented trees.

Let $T = (\mathcal{V}, E, s, d)$ be an ordered oriented tree. For representing the sons v_1, v_2, ..., v_n of a father v, we use exactly n locations. Now, the number of locations of the corresponding vector will be the same as the cardinality of $\mathcal{V}$. Let m be the maximum number of sons (as before). Consider now the array $t : \{1,2,...,s\} \rightarrow (\mathcal{V}, \{1,2,...,m\})$, where $s = |\mathcal{V}|$. The informations contained in t have the following meaning: `t[i]`$= (v, d)$ iff $v \in \mathcal{V}$ and d is the number of its sons (`i` being the position in the depth first visit of T).

Example 6.1.1 *For the tree presented in Example 2.1.1 (Chapter 2), we have:*

```
t[]=(1,3)(2,0)(3,2)(5,2)(10,0)(11,0)(6,0)(4,3)(7,0)(8,0)(9,0)
```

According to the (possible) huge number of "free cells" from the array `t`, the first method for implementing ordered oriented trees, is not convenient for deriving parallel algorithms. The second representation method (although it provides no direct access) allows us to use a bidirectional parsing according to the placement of leaves.

This representation of ordered oriented trees is useful for *depth first search* visits. The input of the following bidirectional traversal algorithm is the array `t[]` (i.e. the corresponding preorder representation of T). In other words, we shall describe an algorithm which use the second method for representing ordered oriented trees. In fact, we present a parallel combination of the two sequential strategies of *up* and *down* traversal. Furthermore, the down traversal coincides with the depth first search strategy. We consider two processors `P1`, `P2` and two global variables `i1`, `i2`. Suppose also that we have a procedure "`halt(P)`", which stops the running of the processor `P`. We shall call this algorithm (BT) (i.e. bidirectional traversal).

We can say that our model is a SIMD (simple instruction stream, multiple data stream) computer ([Akl97]). This means that these two processors `P1` and `P2` operate synchronously. Furthermore, our model is of a *multiprocessor* type because the processors `P1`, `P2` share a common memory.

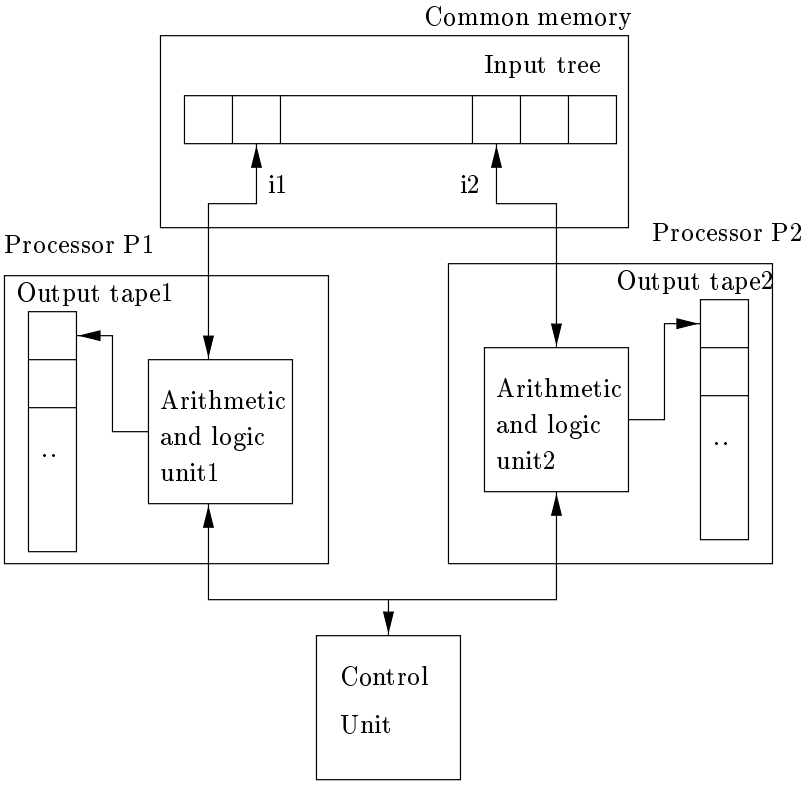


Figure 6.1. General SIMD Model for Bidirectional Traversal

Algorithm (BT) is described using two procedures and a main program:

```
procedure visit_down(P1);
begin
  if i1 <= i2 then begin
    (v1,d) := t[i1];
    {visit the vertex v1}
    write("we have visited ", v1);
    write(" and it has ", d, " sons");
    i1 := i1 + 1;
    visit_down(P1)
  end
  else halt(P1);
end;
```

The up traversal of the tree is quite similar.

```
procedure visit_up(P2);
begin
  if i2 > i1 then begin
```

```
    (v2,d) := t[i2];
    {visit the vertex v2}
    write("we have visited ", v2);
    write(" and it has ", d, " sons");
    i2 := i2 - 1;
    visit_up(P2)
  end
  else halt(P2);
end;
```

The main program is

```
begin
  read(t[]); {read the tree}
  i1 := 1; i2 := n; {n being the number of vertices of T}
  repeat in parallel
    visit_down(P1);
    visit_up(P2)
  until (i1>i2);
end.
```

Example 6.1.2 *Let us reconsider the tree from Examples 2.1.1 and 6.1.1. We shall simulate below the "parallel running" of Algorithm (BT).*

Initial: `i1 = 1` *and* `i2 = 2`.
Step 1: `P1` $\rightarrow$ `we have visited 1 and it has 3 sons`
`P2` $\rightarrow$ `we have visited 9 and it has 0 sons`
Step 2: `P1` $\rightarrow$ `we have visited 2 and it has 0 sons`
`P2` $\rightarrow$ `we have visited 8 and it has 0 sons`
Step 3: `P1` $\rightarrow$ `we have visited 3 and it has 2 sons`
`P2` $\rightarrow$ `we have visited 7 and it has 0 sons`
Step 4: `P1` $\rightarrow$ `we have visited 5 and it has 2 sons`
`P2` $\rightarrow$ `we have visited 4 and it has 3 sons`
Step 5: `P1` $\rightarrow$ `we have visited 10 and it has 0 sons`
`P2` $\rightarrow$ `we have visited 6 and it has 0 sons`
Step 6: `P1` $\rightarrow$ `we have visited 11 and it has 0 sons`
`P2` $\rightarrow$ *halts*

Compared to the classical preorder visit (which needs 11 steps), our bidirectional traversal needs only 6 steps.

We saw in Example 6.1.2 that the processor `P1` visits the tree in depth first search manner (the order of visiting the vertices is $\{1, 2, 3, 5, 10, 11, 6, 4, 7, 8, 9\}$) and `P2` in the opposite manner to `P1` (i.e. the order of visiting the vertices is $\{9, 8, 7, 4, 6, 11, 10, 5, 3, 2, 1\}$). In fact, `P2` visits all the sons from right to left and finally their root.

Theorem 6.1.1 *(correctness and completeness) Let $T = (\mathcal{V}, E, s, d)$ be an ordered oriented tree represented by an array which contains its preorder representation as the input of Algorithm (BT). Then:*

a) After the execution of Algorithm (BT), all the vertices of T have been visited.

b) Let us denote by $T_1(n)$, $T_2(n)$ the running times of the procedures `visit_down` *and* `visit_up`*, where $n = |\mathcal{V}|$. Then the parallel running time $t(n)$ satisfies the relation (we have suppose that the routing time is zero):*

$$\frac{\min\{T_1(n), T_2(n)\}}{2} \leq t(n) \leq \max\{T_1(n), T_2(n)\}$$

Proof

a) If $i1 < i2$ then each call of procedures `visit_down` and `visit_up` implies the visit of two new vertices (which has not yet been visited). We know that the array `t` contains in fact the preorder representation of T. The procedures `visit_down` and `visit_up` read the array `t` cell by cell (because of the statements `i1 := i1 + 1` and `i2 := i2 - 1`). The cells `t[i1]` and `t[i2]` contain informations about the current vertices `v1` and `v2`, respectively. So, `v1`, `v2` have been visited at this parallel step.

b) The inequality $t(n) \leq \max\{T_1(n), T_2(n)\}$ can be obtaining by supposing that one processor waits. For instance, if `P1` waits, 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 implies a running time of $\frac{\min\{T_1(n), T_2(n)\}}{2}$.

■

Remark 6.1.1 *Skipping the details, for constructing the initial tree using the depth first representation, two auxiliary stacks must be available for the two processors.*

6.2 Bidirectional attribute evaluation

In [Alb91b], the *flexible* and the *rigid tree-walking strategies* for traversing the attributed decorated tree have been presented. A *flexible strategy* is completely determined by the attribute dependencies of the grammar concerned. Typical example of attribute grammar classes with a flexible tree traversal strategy are the *absolute non-circular* (ANC) and the *ordered attribute grammars.*

A *rigid strategy* is independent of the attributed dependencies. A typical example of a rigid strategy is to perform a number of passes over the derivation tree, where a *pass* is defined to be a depth-first (left-to-right or right-to-left) traversal of the derivation tree. An example of a strategy somewhere in between the flexible and the rigid strategies is the performance of attribute evaluation

during a sequence of *sweeps* over the derivation tree. A *sweep*, as defined in [EnF82], is a depth-first traversal of the derivation tree, without the restriction of a left-to-right or a right-to-left order of succession, i.e. the visiting order of the nodes is not restricted with the exception that every tree node is visited exactly once.

Our approach refers to a rigid strategy which works for general non-circular attributed grammars. We have called it *bidirectional attribute grammars* strategy. Next, we present the attribute evaluation algorithm which "works" for a well-defined attribute grammar (WAG). We call it *bidirectional attribute evaluation algorithm*, and it will be denoted by (BAE). Each vertex v of the input attributed derivation tree has three components. The first component is the label, i.e. the terminal or non-terminal symbol X, the other components being respectively the set of instances of inherited and synthesized attributes of X.

The Algorithm (BAE):

Input: A well-defined attribute grammar AG and an attributed derivation tree $T = (\mathcal{V}, E)$ where only the inherited attribute instances of the start symbol and the synthesized attribute instances of the terminal symbols are defined.
Output: An attributed derivation tree where all attribute instances are defined.
Method: Like in Algorithm (BT), we shall present two procedures for each of the processors and a main program. Because each element of the array `t[]` stores two informations namely the current vertex and the number of sons of the current vertex (i.e. the pair `(v,d)`), we denote these by `t[].one=v` and `t[].two=d`. Each of them contain two calls of procedure `evaluate(X.a)`. This procedure is in fact the corresponding attribute evaluation instruction of the current production (according to Definition 2.2.3).

```
procedure visit_down(P1);
begin
   if i1 <=|V| then begin
     (v1,d) := t[i1];
     (Xp0,I(Xp0),S(Xp0)):=v;
     for k := 1 to d do
       /* t[i1+1], ..., t[i1+d] are sons of v1 */
       (Xpk,I(Xpk),S(Xpk)) := t[i1+k].one;
     for k := 1 to d do
       for (all a∈I(Xpk)) do
         if (Xpk.a is undefined) and
           (all argument instances of the evaluation instruction
            for Xpk.a are defined)
         then evaluate(Xpk.a);
     for (all a∈S(Xp0)) do
       if (Xp0.a is undefined) and
         (all argument instances of the evaluation instruction
          for Xp0.a are defined)
       then evaluate(Xp0.a);
```

```
    i1 := i1 + 1;
    visit_down(P1)
  end
  else halt(P1);
end;
```

The up traversal of the tree is quite similar.

```
procedure visit_up(P2);
begin
  if i2 >= 1 then begin
    (v2,d) := t[i2];
    (Xp0,I(Xp0),S(Xp0)):=v;
    for k := 1 to d do
      /* t[i2+1], ..., t[i2+d] are sons of v2 */
      (Xpk,I(Xpk),S(Xpk)) := t[i2+k].one;
    for k := 1 to d do
      for (all a∈I(Xpk)) do
        if (Xpk.a is undefined) and
          (all argument instances of the evaluation instruction
           for Xpk.a are defined)
        then evaluate(Xpk.a);
    for (all a∈S(Xp0)) do
      if (Xp0.a is undefined) and
        (all argument instances of the evaluation instruction
         for Xp0.a are defined)
      then evaluate(Xp0.a);
    i2 := i2 - 1;
    visit_up(P2)
  end
  else halt(P2);
end;
```

The main program is

```
begin
  read(t[]); {read the attributed derivation tree}
  i1 := 1; i2 := n;
  {n being the number of vertices of the input tree}
  repeat in parallel
    visit_down(P1);
    visit_up(P2)
  until (all attribute instances are evaluated);
  write(t[]);
  {the attributed derivation tree is consistently decorated}
end.
```

Remark 6.2.1 *Every attribute instance is evaluated during the earliest possible pass and the different instances of the same attribute may be evaluated during different passes. In the classical model of evaluation, the attributed instances of the attributed derivation tree depend in fact on the number of levels (depth) of the input derivation tree. For instance, the synthesized attributed instances of the root of T will be defined after at least m visits of the attributed derivation tree, where m is the depth of it.*

Using our bidirectional attribute evaluation algorithm, the synthesized attribute instances of the root of T will be defined in one visit of the attributed derivation tree, for the synthesized attributed instances of the terminal symbols.

Compared to [Alb91b], our bidirectional attribute evaluation algorithm has "half" time running than the classical attribute evaluation algorithm.

Example 6.2.1 *Let us consider the attribute grammar of Example 2.2.1. It is easy to see that for $w = aab$, according to Algorithm (BAE), we need only* **one** *down visit and* **one** *up visit for evaluating all the attribute instances of the corresponding attributed derivation tree. On the other hand, using classical methods (down visits), for evaluating the attribute instances, we need to visit the attributed derivation tree* **three** *times (i.e. three down visits).*

We can immediately generalize to the input word $w = a^m b$. In that case, Algorithm (BAE) needs the same **one** *down visit and* **one** *up visit, instead of* **m+1** *down visits in the classical method.*

Theorem 6.2.1 *(correctness and finiteness of Algorithm (BAE))*

If the input attributed grammar AG is non-circular, then Algorithm (BAE) will compute the attribute instances of the input attributed derivation tree in a finite number of steps. That is, the output of the Algorithm (BAE) will be a consistently decorated attributed derivation tree.

Proof Let AG be a non-circular (well defined) attribute grammar. This means that for any derivation tree T, the dependency graph $D(T)$ has no cycles. In $D(T)$, the pair $(N_i.a, N_j.b)$ is a directed arc iff attribute instance $N_j.b$ depends on attribute instance $N_i.a$. Because $D(T)$ is acyclic, its arcs specify a partial ordering of the attribute instances. The existence of an arc $(N_i.a, N_j.b)$ indicates that the value of attribute instance $N_i.a$ must be defined before the attribute instance $N_j.b$ can be computed.

The procedures `visit_down(P1)` and `visit_up(P2)` use the procedure `evaluate(X.a)`, which is in fact an attribute evaluation instruction. As a consequence of the use of the loop `repeat ... until` (in the main program) it follows that all the attributed instances of T will be computed. Because $D(T)$ has no cycles, it follows that the number of iterations of the above loop is finite. ■

6.3 Conclusions

The complexity of Algorithm (BAE) applied to an attributed derivation tree is related to the complexity of Algorithm (BT) applied to an ordinary tree. It is

obvious that the number of visits for evaluating all the attribute instances of the attributed derivation tree performed by Algorithm (BAE) is less than that in the classical algorithm ([Alb91b]). As we saw in Example 6.2.1, there exist situations for which our bidirectional evaluating strategy is independent of the size of the input attributed derivation tree.

But, sometimes (due to the dependency graph) the processors have to wait one each other and only one processor works. In that case, the parallel attribute evaluation coincides with the classical one.

If the underlying context free grammar of the corresponding attribute grammar is in *Chomsky normal form*, then any derivation tree will be binary tree. In that case, the first method of representing ordered oriented trees is a convenient data structure for Algorithm (BAE), too. Furthermore, any node of the tree could be directly accessed (i.e. not sequentially, as it was used in the second method of representing ordered oriented trees).

Open problem: In which cases (for what kinds of attribute grammars) our bidirectional attribute evaluation is strictly (or two times) better than the classical one ?

Chapter 7

Final conclusions

7.1 Original contributions of this thesis

The **main original results** are contained in Chapters 3, 4, 5, 6. Our concern was to derive parallel algorithms for parsing and attribute evaluation, at least for some subclasses of context free grammars.

Chapter 3, entitled *Linear-time bidirectional parsing for a subclass of linear grammars*, introduces $LLin(m,n)$ grammars (Definition 3.1.1). They are similar to the $LL(k)$ grammars. Intuitively, instead of only looking ahead to the next k terminal symbols, we have to look ahead to the next m terminal symbols and to look back to the previous n terminal symbols. Thus, the unique production which must be applied can be uniquely determined. Consequently, the membership problem for $LLin(m,n)$ grammars can be solved in linear time. The bidirectional parser attached to a $LLin(m,n)$ grammar is presented in Definition 3.2.2. Theorems 3.2.1, 3.2.2 prove the correctness and termination of the bidirectional parser. Theorem 3.1.5 is due to Prof. Dr. Manfred KUDLEK. $LLin(m,n)$ grammars are treated in detail.

Kernel bibliography: [AnK98, AnK99b, AnG95, JuA97].

Chapters 4 and **5** refer to bidirectional parsing for context free languages. The classical type of parsing is extended by using a mirror process. To analyze an input word, we use two processors acting from both sides of that word. More precisely, the parsing (derivation) tree is in one case (Chapter 4) traversed up to down and left to right combined with down to up and right to left ($LL(k_1) - RL(k_2)$ grammars, Definition 4.4.1). In the other case (Chapter 5), the associated parsing tree is traversed down to up from both sides of the input word using two processors ($LR(k_1) - RL(k_2)$ grammars, Definition 5.3.2). In fact, only one processor will be strongly active after the middle of the input word is found.

Kernel bibliography: [AGK99, AnK99a, AnG95, JuA97].

In **Chapter 7**, a strategy called bidirectional attributed evaluation is emphasizes. To be more precise, this new approach stands for the evaluation of the

attribute instances of an attributed derivation tree. The corresponding (bidirectional) parallel algorithm is also provided.

Kernel bibliography: [AKM99, And97].

7.2 Future work

We intend to:

1. solve the open problems listed at the end of Chapters 3,4,5,6;
2. prove that the power of the bidirectional parsers equals to the power of the Turing machines;
3. find more properties about bidirectional deterministic languages;
4. create and implement a practical compiler generator based on the bidirectional parsing for a two-processor computer (at least for the front-end capability: lexical, syntactical, semantical analysis);
5. even a direct approach for N processors cannot be easily derived, this can be done in a first stage for three and four processors (processes). For instance, we may think that the context free grammars G and $\widetilde{G}$ as being in operator and double Greibach normal form ([Ros67]). Then the point where the initial two processors will meet can be known from the beginning, and one (or two) new processors may be (scalable) introduced.

Bibliography

[AGK99] Andrei, Şt., Grigoraş, Gh., Kudlek, M.: Up-to-Up Bidirectional Parsing for Context Free Languages. *Bericht-221*, Fachbereich Informatik, Universität Hamburg: pp. 1-25 (1999)

[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

[AKM99] Andrei, Şt., Kudlek, M., Masalagiu, C.: Bidirectional Attribute Evaluation. *Bericht-220,* Fachbereich Informatik, Universität Hamburg, pp. 1-16 (1999)

[Alb91a] Alblas, H.: Introduction to Attribute Grammars. *Attribute Grammars, Applications and Systems*, LNCS 545, Eds. Alblas, H., Melichar, B., pp. 1-15 (1991)

[Alb91b] Alblas, H.: Attribute Evaluation Methods. *Attribute Grammars, Applications and Systems*, LNCS 545, Eds. Alblas, H., Melichar, B., pp. 48-113 (1991)

[AlN97] Alblas, N., Nymeyer, A.: *Practice and Principles of Compiler Building with C.* Prentice Hall, U.S.A., 1997

[And97] Andrei, Şt.: Attribute Grammars. *Bericht-202,* Fachbereich Informatik, Universität Hamburg, pp. 1-27 (1997)

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

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

[AnK99a] Andrei, Şt., Kudlek, M.: Bidirectional Parsing for Context Free Languages. *Bericht-219*, Fachbereich Informatik, Universität Hamburg, pp. 1-56 (1999)

[AnK99b] Andrei, Şt., Kudlek, M.: Bidirectional Parsing for Linear Languages. *Developments in Language Theory,* Fourth International Conference, July 6-9, 1999, Aachen, Germany, Preproceedings, W. Thomas Ed., pp. 331-344 (1999)

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, U.S.A., 1986

[Bac79] Backhouse, R.C.: *Syntax of Programming Languages.* Prentice Hall, Englewood Cliffs, NJ, 1979

[Che81] Chebotar, K.S.: Some modifications of Knuth's algorithm for verifying cyclicity of attribute grammars. *Programming and Computer Software* 7, pp. 58-61 (1981)

[Cho56] Chomsky, N.: Three models for the description of languages. *IRE Transactions on Information Theory*, IT-2, pp. 113-124 (1956)

[Cho59] Chomsky, N.: On certain formal properties of grammars. *Information Control*, pp. 137-167 (1959)

[CLR91] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms.* The MIT Press, New York, 1991

[DJL84] Deransart, P., Jourdan, M., Lorho, B.: Speeding up circularity tests for attribute grammars. *Acta Informatica* 21, pp. 375-391 (1984)

[EnF82] Engelfriet, J., Filé, G.: Simple multi-visit attribute grammars. *Journal of Computer and System Science 24*, pp. 283-314 (1982)

[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

[GTW78] Goguen, J. A., Thatcher, J. W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. *Current Trends in Programming Methodology, IV: Data Structuring, (R.T. Yeh Ed.)* Prentice Hall, New Jersey, pp. 80-149 (1978)

[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

[JOR75a] Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. *Comm. ACM* 18, pp. 679-706 (1975)

[JOR75b] Jazayeri, M., Ogden, W.F., Rounds, W.C.: On the complexity of circularity tests for attribute grammars. *Proc. 2nd ACM Symposium on Principles of Programming Languages*, pp. 119-126 (1975)

[Jaz81] Jazayeri, M.: A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars. *Journ. ACM* 28, pp. 715-720 (1981)

[Jou84] Jourdan, M.: Strongly non-circular attribute grammars and their recursive evaluation. *Proc. SIGPLAN'84 Symposium on Compiler Construction*, pp. 81-93 (1984)

[JPJ90] Jourdan, M., Parigot, D., Julié, C., Durin, O., Le Bellec, C.: Design, implementation and evaluation on the FNC-2 attribute grammar system. *Proc. SIGPLAN'90 Symposium on Programming Languages Design and Implementation*, pp. 209-222 (1990)

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

[Kas80] Kastens, U.: Ordered Attributed Grammars. *Acta Informatica*, 13, pp. 229-256 (1980)

[KeW76] Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. *Proc. 3rd ACM Symposium on Principles of Programming Languages*, pp. 32-49 (1976)

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

[Knu68] Knuth, D. E.: Semantics of Context-Free Languages. *Mathematical Systems Theory*, Vol. 2, No. 2 Springer Verlag, New York, pp. 127-145 (1968)

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

[LRS74] Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed Translations. *Journal of Computer and System Sciences* 9, pp. 279-307 (1974)

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

[Lok84] - Loka, R.R.: A note on parallel parsing. *SIGPLAN Notices* 19(1), pp. 57-59 (1984)

[LoP85] Lorho, B., Pair, C.: Algorithms for checking consistency of attribute grammars. *Proving and Disproving Programs, Symposium IRIA*, Rocquencourt, pp. 29-54 (1985)

[Mas92] Masalagiu, C.: *Tipuri abstracte de date*. Editura Universităţii "Al. I. Cuza", Iaşi, 1992

[RaS77] Räihä, K.-L., Saaarinen, M.: An optimization of the alternating semantic evaluator. *Information Processing Letters* 6, pp. 97-100 (1977)

[Ros67] Rosenkrantz, D. J.: Matrix Equations and Normal Forms for Context-Free Grammars. *Journal of the ACM* 14:3, pp. 501-507 (1967)

[RoS97] Rosenberg, G., Salomaa, A.: *Handbook of Formal Languages.* Vol. 1, *Word, Language, Grammar.* Vol. 2, *Linear Modelling: background and application.* Vol. 3, *Beyond words.*, Springer Verlag, 1997

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

[SaS94] - Satta, G., Stock, O.: Bidirectional Context-Free Grammar Parsing for Natural Language Processing. *Artificial Intelligence*, 69, pp. 123-164 (1994)

[Tha67] Thatcher, J. W.: Characterizing derivation trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Science 1*, pp. 317-322 (1967)

[ThS92] Thulasiraman, K., Swamy, M.N.S.: *Graphs: Theory and Algorithms.* John Wiley & Sons, INC., New York, 1992

[TrS85] Tremblay, J., Sorenson, P.G.: *The Theory and Practice of Compiler Writing.* McGraw-Hill, New York, 1985

[TsY77] - Tseytlin, G.E., Yushchenko, E.L.: Several aspects of theory of parametric models of languages and parallel syntactic analysis. In *Methods of Algorithmic Language Implementation*, A. Ershov, C.H.A. Koster (Eds.), LNCS 47, Springer Verlag, Berlin, pp. 231-245 (1977)

[WaC93] Waite, W.M., Carter, L.C.: *An Introduction to Compiler Construction.* Harper-Collins College Publishers, New York, 1993

[WaG84] Waite, W.M., Goos, G.: *Compiler Construction.* Springer Verlag, 1984

[War75] Warren, H.S.: A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations. *Comm. ACM*, Vol. 18, pp 218-220 (1978)

[War62] Warshall, S.: A Theorem on Boolean Matrices. *Journal of the ACM*, Vol. 9, pp. 11-12 (1962)

[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)

[WWY92] - Wu, P.C., Wang, F.J., Young, K.R.: Scanning Regular Languages by Dual Finite Automata. *ACM SIGPLAN Notices*, Vol. 27, No. 4, pp. 12-16 (1992)

Lebenslauf

Name: Ştefan Andrei
Geburtsdatum: 3.04.1970
Geburtsort: Buhuşi, Rumänien
Wohnort: Iaşi, Rumänien
Familienstand: Verheiratet, 1 Tochter

Schulbildung:

1976-1980 Şcoala Generală Nr. 2, Buhuşi, Rumänien, Abt. Grundschule
1980-1984 Şcoala Generală Nr. 2, Buhuşi, Rumänien, Abt. Gymnasium
1984-1988 Liceul Industrial, Buhuşi, Rumänien, Abschluß: Abitur.

Wissenschaftliche Ausbildung und Tätigkeiten

1988 Immatrikulation im Studiengang Mathematik/Informatik der Universität "Al. I. Cuza", Iaşi, Rumänien.

1989-1994 Studium der Informatik and der Universität "Al. I. Cuza", Iaşi, Rumänien. Abschluß: Dipl. Inform.

1994-1995 Master der Informatik and der Universität "Al. I. Cuza", Iaşi, Rumänien. Abschluß: Master Dipl. Inform.

Mai - Juli 1997 DAAD Stipendium an der Universität Hamburg, Arbeitsbereich Theoretische Grundlagen der Informatik.

April - Juni 1998 Tempus Stipendium an der Universität Hamburg, Arbeitsbereich Natürlich-sprachliche Systeme

Oktober 1998 - September 1999 Joint Japan/World Bank Graduate Scholarship Program Stipendium an der Universität Hamburg, Arbeitsbereich Theoretische Grundlagen der Informatik.

1995-1999 Veröffentlichungen von 16 Artikeln (6 in Internationalen Zeitschriften, 3 in Rumänischen Zeitschritfen, 7 Berichte), 5 Sommerschulen, 2 Konferenzen